Debreceni Egyetem - Informatikai Kar

Web – szerver és mobil – kliens alkalmazások fejlesztése

Témavezető:

Dr. Fazekas Gábor

Készítette:

Budai Attila és **Csubák Péter**

2009
Debrecen

Tartalomjegyzék:

Bevezető

A szakdolgozat célja megismertetni a programozóval azokat az alapfogalmakat amelyek szükségesek a web – szerver és mobil – kliens alkalmazások fejlesztéséhez. Továbbá szeretnénk bemutatni technológiákat, eszközöket, programozási nyelveket, melyek az ilyen irányú fejlesztések esetén használatosak. Ezeket természetesen nem áll módunkban teljes mivoltjukban bemutatni, de említésre kerülnek mint alternatívák. Ezen eszközök közül egyet – egyet mindkét oldalon: mind szerver-, mind kliensoldalon szeretnénk kiemelni és bemutatni. Mindezt pedig egy konkrét példán keresztül tesszük meg, azaz azt mutatjuk be, hogy hogyan is kell megírni egy ilyen alkalmazást. Mivel nem áll módunkban minden apró részletre kitérni, sőt gyakorlatilag az általunk megemlített témák mindegyikéből több tucat szakdolgozatot lehetne írni, így feltételezzük az olvasóról, hogy valamilyen szinten ismeri az általunk használt nyelveket, eszközöket. Még megemlítjük, hogy az általunk használt eszközök, ami az operációs rendszertől kezdve, a tervezőeszközökön át, a legkisebb felhasznált modulig mind – mind ingyenesek. A Linux/Unix rendszereket részesítjük előnyben, de a legtöbb általunk használt eszköz elérhető Microsoft Windows platformra is. Fontos még megemlítenünk, hogy néhány, az általunk felhasznált eszközök működési elvével nem feltétlen értünk egyet, viszont a hibáikat elfogadva – , néhol javítva – használjuk ezeket.

Szerver-kliens alkalmazással a mindennapi életünkben is találkozhatunk: gondoljunk csak akár az e-mail kliensekre, amelyek az e-mail szolgáltatók szerveréhez kapcsolódva hajtanak végre műveleteket azon. Valami hasonlóra kell gondolni, ha meg akarjuk érteni, mit is jelent a web-szerver és mobil-kliens alkalmazások megnevezés, hiszen az iménti példától ez annyiban tér el, hogy a kliens oldalon nem egy PC, hanem egy mobiltelefon van. Természetesen a lényeg a szoftverben van, tehát nem az számít, hogy milyen hardvereszközök találkoznak a szerver illetve a kliens oldalon, hanem hogy az azokon futtatott szoftver felépítése milyen, azaz hogy mi a célja, és azt hogyan is valósítjuk meg ilyen környezetben. Természetesen a hardver befolyásolja a szoftvert is, hiszen egyértelmű, hogy a mobiltelefonok teljesítménye nem ér még fel egy átlagos PC teljesítményével, viszont még inkább befolyásolja a kommunikáció mennyiségét és milyenségét. Ezért dolgozatunk hangsúlyt helyez ezen tényezőkre is. Tehát azt is szeretnénk bemutatni, hogy a hardver, szoftver, és a kommunikáció terén milyen módokon

található meg a harmónia egy-egy ilyen alkalmazás elkészítése során.

Mit is jelent a Web – szerver illetve mobil – kliens alkalmazás? Olyan alkalmazások együttese, amelyek kliens oldalán található egy mobil eszköz, amelyen tetszőleges alkalmazás fut, előre definiált céllal, és hogy célját elérje szüksége van információkra/funkcionalitásra, amelyet egy, vagy akár több kiszolgáló juttat el hozzá a világhálón keresztül.

Sajnos egyelőre nem létezik egységes megoldás, amelynek segítségével megoldható lenne az ilyen szoftverek transzparens fejlesztése (bár a Sun Microsystems erősen próbálja kiterjeszteni mindenre a Java nyelvet). Éppen ezért aki ilyen fejlesztésbe vág, az először azzal a problémával fogja szembetalálni magát, hogy mind a szerver, mind a kliens oldalon nagyon sok, és nagyon eltérő programozási nyelv létezik és ezek közt nem nagyon akad olyan, ami a kliens és a szerver oldalon is megtalálható. Így a kommunikációhoz is újabb strukturált nyelvekre van szükség, melyek megteremtik a kompatibilitást. Tehát az első feladat a fejlesztés alapját jelentő programozási nyelvek áttekintése. Így mi is ezzel kezdjük.

A programozási nyelvek amelyekről szólunk a teljesség igénye nélkül: J2ME, Python, Javascript, XML, YAML, PHP, JEE. Ezek közül kiemeljük azokat, amelyeket felhasználunk a példánkban (és azokat, amelyekről szót ejtünk, mint fontosabb alternatívák):

- Kliens: J2ME, Javascript, (Python, C++)
- Szerver: PHP, (JEE)
- Kommunikáció: XML, (YAML)

A megfelelő nyelvek megválasztása után a második legfontosabb lépés a tervezés, melyhez szerencsére egy nagyon kiforrott megoldás létezik: ez az UML. Az UML-ről feltételezzük, hogy az olvasó már ismeri, így ennek részletesebb bemutatásától eltekintünk. Minthogy ez a legmegfelelőbb eszköz egy alkalmazás megtervezéséhez, így mi is ennek segítségével mutatjuk be példa alkalmazásunk terveit.

A példa feladatról:

A mobil DVD kölcsönző egy olyan Web-szerver mobil-kliens alapú háromrétegű alkalmazás, ami egy olyan online elérhető DVD kölcsönző, amelynek segítségével online lehet a DVD kölcsönzéseket menedzselni mind a front-end, mind a back-end oldalon. Tehát a rendszer feladata egyrészről kiszolgálni a megrendelőket, akik egy publikus

weboldalon keresztül válogathatnak a DVD-k közt, állíthatnak össze egy rendelési listát, regisztrálhatnak, illetve bejelentkezés után lehetőségük van megrendelni az általuk kiválasztott DVD-ket. Másrészről ki kell szolgálnia a mobil klienssel a rendszerhez kapcsolódó futárokat, akik a megrendelt DVD-ket szállítják ki a megrendelőnek, és ha a megrendelőnél vannak visszaküldendő DVD-k, akkor azokat vissza is szállítják a raktárra.

Kliens oldali nyelvek áttekintése:

Amikor kliens oldali nyelvet választunk, a fontos szempontokat kell megvizsgálnunk:

- A telefon milyen nyelveken programozható?
- Ezeken a nyelveken milyen lehetőségek vannak grafikus megjelenítésre?
- Milyen támogatottságot biztosít a hálózatkezeléshez, esetleg az XML feldolgozáshoz?

Symbian C++

A C++ nyelv alapjait Bjarne Stroustrup fektette le. (1983-1985). A C++ kiterjesztése a C nyelvnek. Bjarne Stroustrup bizonyos tulajdonságokkal bővítette a C nyelvet, amelyeket formalizált, és „C with Classes” (C nyelv osztályokkal) - nak nevezett. Kombinálta a Simula osztályait és az objektum orientáltság sajátosságait a hatékony C nyelvvel. A C++ kifejezést 1983-ban használták először. Ez volt az alapja a C++ programozási nyelvnek. Mára már az egyik leghatékonyabb programozási nyelvvé nőtte ki magát. Világszerte fejlesztenek ezen a nyelven, és fejlesztik magát a nyelvet is.

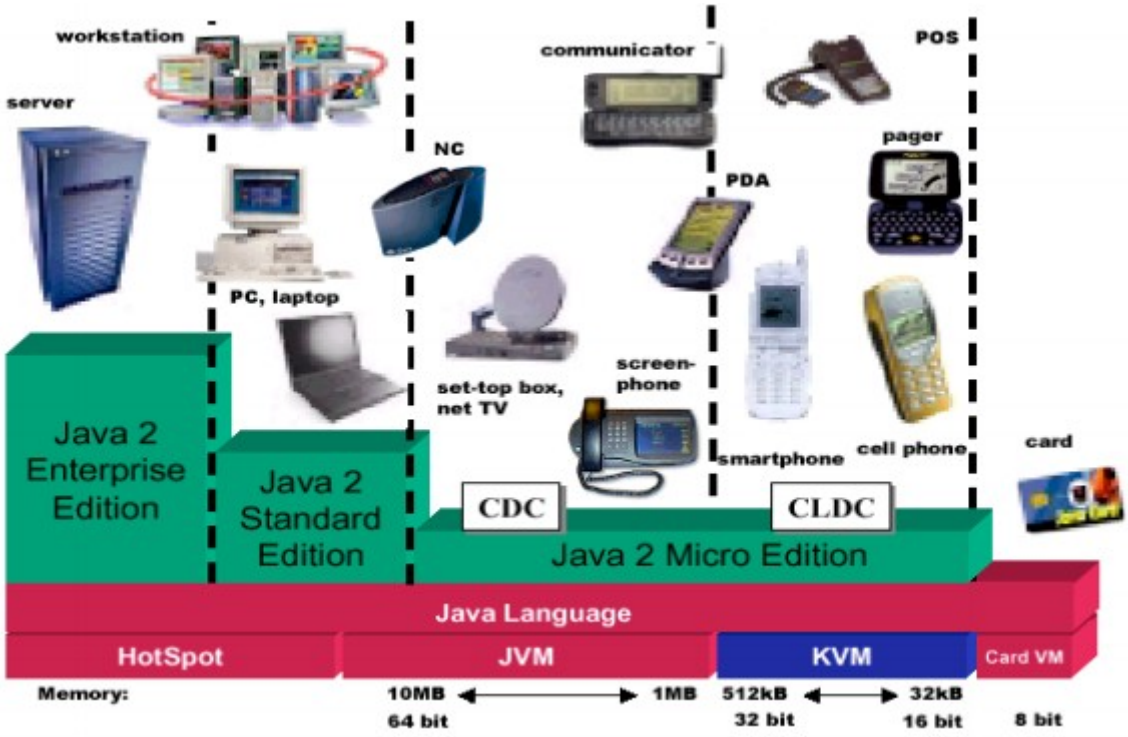
Ahhoz, hogy összeköttetésbe hozzuk a mobilprogramozással, szükséges további információkat ismertetnünk. Elsősorban fontos néhány dolgot tudnunk a Symbian OS-ről, amely a mobil-operációs rendszerek egyik legjelentősebb képviselője (jelenleg a 9.5-ös verziónál tart) . Kezdetétől fogva kis erőforrás igényű, magas rendelkezésre állású, biztonságos rendszernek tervezték. A kernelt, magasabb szintű szolgáltatásokat tartalmazó rétegeket C++ -ban írták. (Tehát erre az operációs rendszerre natív alkalmazásokat C++ -ban lehet írni.) A mobil operációs rendszereknek ez csak az alapját képezi, erre ráépül egy felhasználói felület (UI) réteg, amelyek eléggé különbözőek lehetnek. Kezdetekben az eltérő eszközökhöz különböző keretrendszereket definiáltak az eltérő képernyőméret vagy a különböző beviteli eszközök függvényében. Ezek a keretrendszerek, amelyek alap alkalmazásokat is tartalmaztak, később komplett szoftverplatformokká fejlődtek. (A mai két legjelentősebb szoftverplatform: S60, UIQ.)

Azért szerepel az alcímben Symbian C++, mert gyakran így nevezik a Symbian OS programozásakor használt C++ nyelvet. Ennek az az oka, hogy kezdetekben, a Symbian OS kezdeteiben, a C++ sem volt régi nyelv, így a nyelv sok része, amelyek természetesen mára már kiforrottak, még definiálatlanok voltak. Ennek a következménye a Symbian OS

alkotói sok egyedi, sok megkötéssel járó technikát vezettek be, és ezek olyan mélyen épültek be az operációs rendszerbe, hogy még manapság is alkalmazkodni kell hozzájuk. De ne értsük félre, ezek nem rosszak, legtöbbjük optimális processzor- és memóriakihasználtságot biztosítanak, viszont megnehezítik a szabványos C++ -hoz szokott programozók dolgát. Azért azt még megjegyezzük, hogy Symbian OS – re fejleszthetünk Python nyelven, és Java ME - ben is, de természetesen legnagyobb szabadsága a programozóknak C++ -ban van, mert elérhetjük a beépített alkalmazások szolgáltatásait, felhasználói felület teljesen hozzáférhetővé válik, és még sorolhatnánk.

J2ME

A Java Micro Edition valójában nem más, mint Java programozási nyelv, amely segítségével „kis” készülékeket/beágyazott eszközöket (Mobil telefonokat, PDA -kat, stb.) programozhatunk Java nyelven, de nem a jól megszokott SE (Standard Edition) vagy a még robusztusabb EE (Enterprise Edition) eszközök felhasználásával (a tisztánlátás kedvéért: 1. kép). Természetesen a J2ME is szabványosított API (Application Programming Interface) -val rendelkezik.



1. Kép

A mobileszközök teljesítménye, igencsak korlátozott, mind számítási teljesítmény-

, mind memória- , mind pedig rendelkezésre álló tárkapacitás terén, és nyilván látható a legegyszerűbb példán is, hogy egy többablakos desktop alkalmazás nem fog futni egy mobilkészüléken. Ezért szükséges volt egy olyan futtatókörnyezetre, amely képes mobileszközök korlátozottságai között futtatni a Java kódjainkat. Ezt a célt szolgálja valójában a J2ME.

A nyelv ismerete, filozófiájának ismerete természetesen nem elég ahhoz, hogy programot írjunk mobileszközre, igaz van kompatibilitás az SE és ME osztályai között, de további ismeretek szükségesek. A legfontosabb az architektúra ismerete, valamint a különbségek a Standard Editionhoz képest, de minderről, és a fontosabbakról később részletesebb leírást adunk.

Manapság szinte alig készül olyan mobiltelefon amelyet ne tudnánk programozni Java nyelven. Valójában ez az egyik fő oka, hogy ezt a nyelvet választottuk, a másik oka az a nyelv lehetőségeiben rejlik. A mai világ egyik leggyorsabban fejlődő nyelve a Java, és nem kivétel ez alól a Micro Edition része sem.

Java ME esetén a különböző opcionális API-kban található meg a támogatottság a GUI-ra nézve. Több eszközrendszerről van szó és amennyiben a telefonunk támogatja, akkor használhatjuk ezeket. Természetesen van alapértelmezett GUI eszközrendszerünk is.

Az MIDP 1 -hez eléggé kezdetleges tartozik, viszont amikor készült, akkor minden igényt kielégített. A MIDP 2 már bőségesebb kínálattal rendelkezik, ez már a fejlettebb telefonokhoz készült. A legújabb az MIDP 3-hoz tartozóak, természetesen újabb fejlesztések kerültek bele. Igaz még nem található meg minden ember kezében olyan telefon, amely támogatja, de már léteznek ilyen telefonok is.

A legfontosabb része a grafikus megjelenítésnek Java ME-ben, hogy mi is implementálhatjuk a megfelelő interfészt, és saját megjelenítő felületet hozhatunk létre (pl: FIRE). Léteznek még további eszközök, például a JSR - 226-ra keresztelt opcionális API, amely SVG - Scalable Vector Graphics (tiny) támogatottságot jelent. Ezzel könnyű animált menüket, mozgó képeket a kijelzőre varázsolni.

A kommunikációhoz eszközöket szolgáltat a Java ME, legyen szó HTTP kapcsolatról, HTTPS-ről vagy akár TCP/IP kapcsolatról. Igaz, a java ME kapcsolatkezelésének tervezésekor az SE-ben használt java.io és java.net csomagok túlságosan erőforrás igényesnek bizonyultak, így egy teljesen új keretrendszert alkottak meg. Ez lett a GCF (Generic Connection Framework). A GCF osztályok a MIDP-t támogató készülékeken futtathatóak.

Az XML kezelésére több eszközt is szolgáltat a Java ME. XML feldolgozásra az egyik a JSR-172, vagyis a JAXP (-Java API for XML Processing) amelynek a javax.xml.parsers csomagja tartalmazza a SAX (Simple API For XML) elemzőt. A másik a kXML elemző, amelyet az MIDP számára terveztek, ideális választás XML feldolgozásra, nagy XML fájlok esetén is hatékonyan használja az erőforrásokat.

Python

Ez a nyelv egy általános célú, magas szintű, dinamikus, objektumorientált programozási nyelv. Nagy előnye, hogy támogatja a procedurális paradigmát is. Általában a szkriptnyelvek családjába sorolják, pedig valójában jóval több annál, sokkal többet kínál mint az általános script nyelvek, batch fájl–ok. Erről árulkodik a nagyon sok funkcióval ellátott Standard könyvtára, amely tartalmaz például adatbázis kapcsolatokat, kommunikációs protokollokat (http), fájlkezelést, rendszerhívásokat támogató szolgáltatásokat, támogat magas szintű típusokat (pl: lista), valamint rendkívül sok beépített eljárást és csomagot tartalmaz, rengetek újrahasznosítható, és teljes mértékben Python nyelven írt modul található benne, amelyekkel természetesen gyorsabban fejleszthetjük alkalmazásainkat. Ezeken túlmenően könnyen illeszthető, jól bővíthető C++ -ban, vagy C -ben megírt modulokkal.

Python futtatókörnyezet számtalan különféle rendszerre létezik például: Unix -ra, MacOS X -re, iPhone -ra, Palm készülékre, és az imént említett Symbian OS készülékekre is, valamint Microsoft Windows-ra is. Elsősorban kliensszoftverek, prototípusok készítésére alkalmas, mégis annak az oka, hogy mi nem ezt használjuk az az, hogy a legtöbb mobilkészülék nem tartalmazza a Python futtatókörnyezetet.

A jelenlegi legnépszerűbb felhasználói felület a Symbian operációs rendszerre épül (az S60-ra és az erre az operációs rendszerre elkészített Python-implementációnak a neve PyS60). A PyS60-as alkalmazások felhasználói felülete nem tér el az S60 készülékek által nyújtott GUI-tól. E célból a PyS60 egy olyan modult , az *appuifw*-t kínálja, amely az S60 grafikus felületére képezi le a felhasználói interakciókat végző függvényeket.

Szerver oldali nyelvek áttekintése:

Szerver oldali nyelvek választásakor fontos kérdések, hogy milyen eszközöket biztosít:

- a kommunikációhoz
- perzisztencia kezeléshez
- XML kezeléshez

JEE

A Java Enterprise Edition-t először 1998-ban mutatta be a SUN Microsystems, akkor még JPE (Java Platform, Enterprise Edition) néven, gyakorlatilag ez tekinthető az 1.0 verziónak. A standard Java fejlődésének egy fontos pontja volt az 1.2-es verzió, sokan akkor Java2-nek is titulálták azt, ennek megjelenésekor (1999 December) az Enterprise verzió is megkapta a J2EE (Java 2 Enterprise Edititon, vagy Java Platform 2, Enterprise Edition) nevet. A standard Java-tól az EE annyiban különbözik, hogy rengeteg API-t hoz magával, amelyek az üzleti folyamatok megkönnyítését szolgálják (illetve hogy egy szerverkörnyezetet biztosít a Java alapú alkalmazásoknak), az 1.2-es verzióban ezen API-k bővültek/finomodtak leginkább (EJB 1.1, JSP 1.1, Java Servlet 2.2, JDBC 2.0, JNDI 1.2, JMS 1.0.2, JTA 1.0), illetve az EE is megkapta a standard Java 1.2-ben bemutatott új nyelvi elemeit. A Java Enterprise Edition egészen az 1.5-ös verzió megjelenéséig (2006) J2EE néven volt ismert, ami részben utalt arra is, hogy az 1.2-ben bevezetett szabványokkal, és API-kkal kompatibilis verzióról van szó, illetve utalt arra is, hogy a fent említett API-k nem változtak drasztikusan. Az 1.5-ös standard Java-ban viszont behoztak pár új nyelvi elemet (annotációk, foreach ciklus, generikus) amelyek mentén ezeket az API-kat is jelentősen át lehetett tervezni a gyorsabb és kényelmesebb fejlesztés érdekében. Így a SUN Microsystems át is nevezte a J2EE-t JEE-re (Java Enterprise Edition) illetve JEE5-re (Java Enterprise Edition version 5). A JEE5-ben bevezetett újítások közül fontos kiemelni az EJB 3.0-t (Enterprise Java Bean) amely nem kompatibilis a régebbi 2.0-ás verzióval, (természetesen ettől a JEE5-ben van támogatás a 2.0 verzióhoz is) mivel megszüntette az Entity Bean-eket, helyettük szimpla egyszerű POJO-kban (Plain

Old Java Object) definiálhatóak az entitások, vagy xml leíró segítségével, vagy annotációkkal. Ezzel jelentősen felgyorsították a fejlesztési folyamatokat, és megszüntették az értelmetlen osztály-interfész duplikációkat. Természetesen sok más API is átesett néhány nem is kevés átalakításon, aminek keretében az új nyelvi elemek segítségével jóval egyszerűbbé tették a fejlesztést. Ezen dolgozat írásakor a legfrissebb JEE verzió az 1.6-os viszont mivel ez még nagyon új, és nem hozott akkora mennyiségű újítást, hogy egy ilyen rövidebb bevezető keretében azokról is beszélni lehessen, így többször az 1.5-ős (JEE5) verzióra fogunk hivatkozni.

A Java Enterprise Edition tulajdonképpen a Java alapú szerverprogramozás eszköze. Maga a nyelv ami a programozásához használható, az a Java Standard Edition 1.5-ös verzió nyelvi specifikációjával kompatibilis.

Maga a JEE tulajdonképpen egy eszköztár, egy nagyon komplex eszköztár, amivel egy úgynevezett alkalmazás szerverre (Application Server) lehet szoftvereket fejleszteni. Ilyen alkalmazás szervert ad maga a SUN Microsystems is, ez nem egyezik meg a JVM-el (Java Virtual Machine), a JVM tulajdonképpen ez alatt helyezkedik el, és csak a konkrét nyelvi utasításokat "interpretálja", magát az alkalmazásszervert már a szoftver és a futtatókörnyezet vezérli. Az alkalmazásszerverek egy specifikáció mentén épülnek fel, amit maga a SUN Microsystems állított össze, és ezt a specifikációt implementálják az alkalmazásszerverek. Az alkalmazás szerver feladata hasonló az adatbázis szerveréhez, az összeköttetést adja meg az alkalmazásunknak. Az összeköttetést a kliens alkalmazással (legyen az bármilyen kliens is), összeköttetést a szerver (a számítógép) szolgáltatásaival, és természetesen összeköttetést a különböző erőforrásokkal (adatbázisok, tartalomkiszolgálók, protokollok, ...stb.)

Alkalmazás szervert sok cég gyárt, alapvetően a rengeteg szabvány miatt ezek közt olyan nagy különbség nem érezhető, viszont komolyabb fejlesztések során erőteljesen kijöhetnek egyes alkalmazás szerverek gyengeségei. Éppen ezért nem árt tisztában lenni az alternatívákkal. A legismertebb alkalmazás szerverek:

- Glassfish (SUN - ingyenes nyílt forrású referencia implementáció)
- IBM WebSphere (Verziótól függő fix árazású)
- Oracle Weblogic (Eredetileg a BEA fejlesztette, amit felvásárolt az Oracle, processzor alapú licencelésű)

- Oracle Application Server 10g (Oracle saját fejlesztése)
- JBoss (RedHat nyílt forrású alkalmazás szerver, csak a support, illetve az oktatás kerül pénzbe)
- Adobe JRun (Processzor alapú licencelésű)
- Apache Geronimo (Nyílt forrású)
- SAP Netweaver
- Apple WebObjects

A JEE kapcsán sok mindennel lehetne foglalkozni, minthogy annyira kiterjedt és szerteágazó eszközrendszer, de az egyik legfontosabb kérdés a szerveren futó enterprise alkalmazásokkal kapcsolatban, hogy hogyan oldódik meg a perzisztencia kérdése. A bevezető elején említettük, hogy sok programozási nyelvet kell átlátni, és ismerni annak aki ilyen fejlesztésbe kezd, de ott nem véletlenül nem említettük az adatbázis vezérlő nyelveket. Ennek az oka, hogy a szerveroldalon a perzisztencia megoldása (főleg JEE-ben) inkább a perzisztencia kezelő rendszerek dolga, amelyek mondhatni függetlenek az adatbázisrendszerektől. Így gyakorlatilag folyamatosan objektumokkal dolgozhatunk, nincs szükség rá, hogy ismerjük az adatbázis-vezérlő nyelvet. Ezek a rendszerek mintegy elrejtik előlünk az adatbázist és annak vezérlését, és mindent magas szinten tudunk vezérelni. JEE-ben az API-ban is találunk ilyen eszközöket: JDBC, EJB-Entity, de ilyen eszközöket külső forrásból is be lehet szerezni, a legismertebb ilyen független eszközrendszer a Hibernate.

A másik fontos kérdés a kommunikáció. Alapvetően igaz az, hogy XML alapú kommunikációra építünk, ehhez ad kiterjedt eszközrendszert a JEE. A sokféle XML alapú kommunikáció közül a mobil kliens miatt viszont egyedül az XML-RPC marad mint használható alternatíva. Viszont költséghatékonyabb és gyorsabb megoldás lehet, ha az XML üzeneteket nem ilyen magas szintű eszközökre bízzuk, hanem magunk tervezzük meg. Ehhez is nagyon jó eszközrendszereket biztosít a Java már az SE-ben is amiket természetesen a JEE is örököl. A kommunikáció kapcsán a másik fontosabb kérdés a célba juttatás. Ehhez a legegyszerűbb a Web-alkalmazások fejlesztéséhez használt megoldás. Azaz, hogy a választ egy weblapként generálja a szerver, a kérés pedig a szabványos böngészőkben is használt POST módszerrel érkezik a szerverre. Ebben az esetben természetesen egyszerűsíthető a kommunikáció, ha a POST által engedélyezett

változókban küldjük az adatokat a szerverre, viszont az XML alapú kommunikáció egységesebb, és például az objektumok üzenetté alakítására nem kell saját megoldásokat kialakítani.

Érdemes még megemlíteni, hogy ha a kliens Javascript alapú, akkor a kommunikációt lehet egyszerűsíteni, és esetleg az XML alapú kommunikációt HTML alapúra lehet cserélni. (Ennek természetesen feltétele a kliens asszinkron kommunikációt biztosító képessége, okos telefonoknál ez szinte már követelmény is).

Fontos megjegyezni, hogy sok gyártó előre gyártott megoldásokat szolgáltat a mobil eszközök kiszolgálására, például az Oracle a Fusion Middleware csomagjában található ADF Faces API-ban ad lehetőséget a mobil kliensek kiszolgálására. Viszont ez a többségüknél kimerül abban, hogy a normál JSF oldal helyett egy olyat tervezhetünk, ami a mobil eszköz böngészőjében is megjelenhet. Ez önmagában még nem lenne probléma, de sok eszköz még nem támogatja, és általában az ilyen előre gyártott mobil megoldások sem támogatják az asszinkron kommunikációt, aminek az a nagy hátránya, hogy a kliens oldalon egyáltalán nem költséghatékony a megoldás. Ennek fő oka, hogy a kommunikáció nem csak a hasznos információra korlátozódik, hanem a megjelenítési információt is magában hordozza, és egyetlen egy művelet esetén minden megjelenítési információnak is el kell jutni a mobil eszközre. És mivel ez a kommunikáció tipikusan a mobilhálózatokon keresztül folyik így ez jelentősen emeli az alkalmazás használatának költségeit.

Lehetne még beszélni nagyon sok mindenről a JEE kapcsán, de egyelőre elég ezeket a szempontokat figyelembe venni. Azt kell meglátni még a JEE-vel kapcsolatban, hogy a JEE eszközrendszert komplex nagyvállalati megoldásokra szánták, ennek fényében, kevésbé használható a mobil eszközök területén. Nem mintha a nagyvállalatok nem dolgoznának mobiltelefonokkal, vagy nem lenne ilyen rendszerekre szükségük, a probléma inkább az, hogy a nagyon komplex feladatok, az optimalizált, nagyon leegyszerűsített kommunikáció, és a gyengébb kliensoldali képességek (főleg ami az absztrakciós szintet illeti) nehezen egyeztethetőek össze. Annál is inkább, hogy a JEE sokkal inkább abba az irányba tart, hogy a fejlesztés legyen minél gyorsabb, és minél egyszerűbb, nem számít ennek milyen teljesítménycsökkenés az ára. Mindez a mai több gigahertzes, többmagos processzorok és a több gigabyte memóriák korában nem is számít a PC-s világban, de igenis számít a mobil eszközök világában.

PHP

A példa alkalmazás készítéséhez ezt a nyelvet fogjuk használni, így itt most csak a JEE-nél tárgyalt szempontokból fogjuk megnézni a PHP-t.

A PHP egy programozási nyelv, teljesen nyílt forráskódú, akárcsak a Java. Viszont a PHP kifejezetten a webszerverekre készült script nyelvként kezdte a pályafutását 1995-ben Rasmus Lerdorf keze nyomán aki eredetileg a saját weblapján futtatott Perl scripteket akarta lecserélni. Ennek megoldására írt egy eszközrendszert amit elnevezett "Personal Home Page tool"-nak azaz PHP-nak. Mára a PHP elérte az 5-ös verziót (jelenleg az 5.3-as verziónál tart, de már létezik működőképes 6-os fejlesztői verzió is) és komoly alkalmazásfejlesztői rendszerré nőtte ki magát. Ezt fogjuk is példázni a későbbiekben. Mindezek mellett viszont megmaradt az ami mindig is volt, azaz webszerverek script nyelve. A PHP azon túl, hogy egy programozási nyelv, egyúttal egy futtatórendszer is. Hasonlónak lehet tekinteni, mint a Java JVM-ét, csak ebben az esetben nincs szükség konkrét alkalmazás szerverre a működéshez. Viszont szükség van egy Webszerver alkalmazásra, ami a kiszolgálni kívánt oldalakban a PHP futtatókörnyezetének segítségével futtatja a PHP kódot, és a végeredményt szolgálja ki. Ebből a szempontból a PHP gyakorlatilag olyan mint minden más szerveroldali script nyelv.

A PHP legnagyobb előnye az, hogy mindamellett, hogy megtartotta az egyszerű vonásait, amit mint script nyelv kellett teljesítenie, egy nagyon komplex rendszerré vált, amiben nagyon komoly fejlesztések is könnyen megvalósíthatóak. Bár perzisztencia kezelő eszközöket olyan magas szinten nem tartalmaz mint a JEE, de egyrészt nagyon kényelmes eszközrendszert kínál az adatbázis kezelésre, másrészt könnyen fejleszthető benne saját perzisztencia kezelő rendszer - egy ilyen rendszer felépítését be is mutatjuk a későbbiekben, ami tulajdonképpen része annak a saját fejlesztésű rendszernek amiben a minta alkalmazás szerver oldali részét fejleszteni fogjuk.

A kommunikáció a PHP esetén is felmerül, és szerencsére itt is léteznek nem túl magas szintű, de mégis nagyon könnyen használható XML feldolgozó eszközök. A kommunikáció útja pedig hasonlatos a JEE esetében ismertetettel, azaz, hogy a választ mint egy weboldalt generáljuk, a kérés pedig a POST "metódussal" történik. Ez a

legegyszerűbb, és legkézenfekvőbb megoldás. Érdemes még megemlíteni, hogy a PHP-ban lehetőség adódik egy másik üzenetformátum használatára, mégpedig YAML nyelven. Ez a nyelv a mobil kliensek szempontjából az ideálisabb választás, mert jóval kevesebb a típusleírás benne, mint az XML-ben, és sok esetben bizonyos struktúrák jóval könnyebben írhatóak le vele. A probléma a YAML-el, hogy komplikáltabb, mint az XML, és bár adatforgalom szempontjából kedvezőbb, a kevésbé egységes szerkezet miatt nehezebben használható.

Kommunikáció:

XML

Már olvashattunk pár szót az XML-ről, de most kicsit bővebben annak a tükrében, hogy az olvasó már hallott és használt XML -t.

Maga az XML 1998-ban jelent meg a W3C hivatalos specifikációjaként. A Java platformfüggetlen kód írásának lehetőségét teremtette meg, az XML pedig a platformfüggetlen adatábrázolást és adatcserét teszi lehetővé, ezért a kettő nagyon erős párost alkot. A jelölő nyelveket (XML – Extensible Markup Language) a dokumentum jelölésekkel való ellátására használjuk, amelyek a későbbi feldolgozást segítik. XML hordozhatóságról nincs nagyon értelme beszélni DTD (Document Type Declaration – dokumentum típus meghatározás) nélkül. Gyakorlatilag a dokumentum „nyelvtanát” adjuk meg vele, azaz, hogy milyen elemeket használhatunk és azok hogyan viszonyulnak egymáshoz, illetve milyen attribútumaik lehetnek. Ahogy az XML egyre elterjedtebb lett, úgy lett egyre világosabb hogy a DTD nem elégít ki minden igényt, és a vele szemben támasztott követelményeket. Sok dolgot csak nagyon körülményesen lehet vele elérni és vannak olyan dolgok, amelyeket nem is tesz lehetővé. Másik probléma vele, hogy az XML mellett ismerni kellett egy másik leíró nyelvet, mivel a DTD nem az XML-ben volt. Így a W3C elkészítette az XML séma specifikációt. Ennek lényege, hogy a sémát, magában az XML-ben kell megírni. Ugyan van DTD-je az XML sémának is, viszont ezzel már a programozónak nem kell foglalkoznia (hasonlóan mint a HTML-nél).

Az XML széleskörű használatát jól mutatja még, hogy szabványok léteznek arra, hogyan lehet egy XML dokumentumot egy másik fajta XML dokumentummá transzformálni (XSLT – Extensible Stylesheet Language Transformation), vagy olyan bináris formára hozni mint pl. egy PDF fájl, vagy ODT fájl. Vagyis szabványok léteznek arra, hogy XML formában tárolt adatokból WML vagy XHTML dokumentumot hozzunk létre.

Szintén a W3C ajánlása a DOM (Document Object Model) API is. Ez valójában egy objektum modell az XML dokumentumok ábrázolása. Arról van szó, hogy fa adatszerkezetként használjuk magát az XML–t. A DOM felépíti ezt az adatszerkezetet a memóriába az XML-ből, és ezek után bánhatunk vele faként.

Fontos még megemlíteni, hogy manapság tényleg e körül forog a világ. Gyakorlatilag nincs olyan terület, ahol ne használnánk XML-eket. A legnagyobb

alkalmazásszerverek (IBM WebSphere, Jboss, WebLogic, Glassfish) mind ezt használják, nem beszélve az Apache Ant-ról, a Maven-ről, és még sorolhatnánk. Mindezek miatt is feltételezzük, hogy az olvasó már hallott róla.

Mobil – kliens (J2ME) esetében is vannak különböző parser-ek. Az általunk favorizált a kXML csomag (a MIDP-hez készítették, de nem hozzá, viszont ingyenesen letölthető, használható). Mivel az általunk használt XML struktúrája nem lesz a legkomplikáltabb, így elég egyszerű parser-re van szükségünk, amit akár könnyen implementálhatunk mi is (az alkalmazás mérete ugyanis nagyon korlátozott egyes telefonoknál), de nyilván ez a XML komplexitásából fog adódni, hogy mit is érdemesebb használni.

Fejlesztői környezetek (IDE-k):

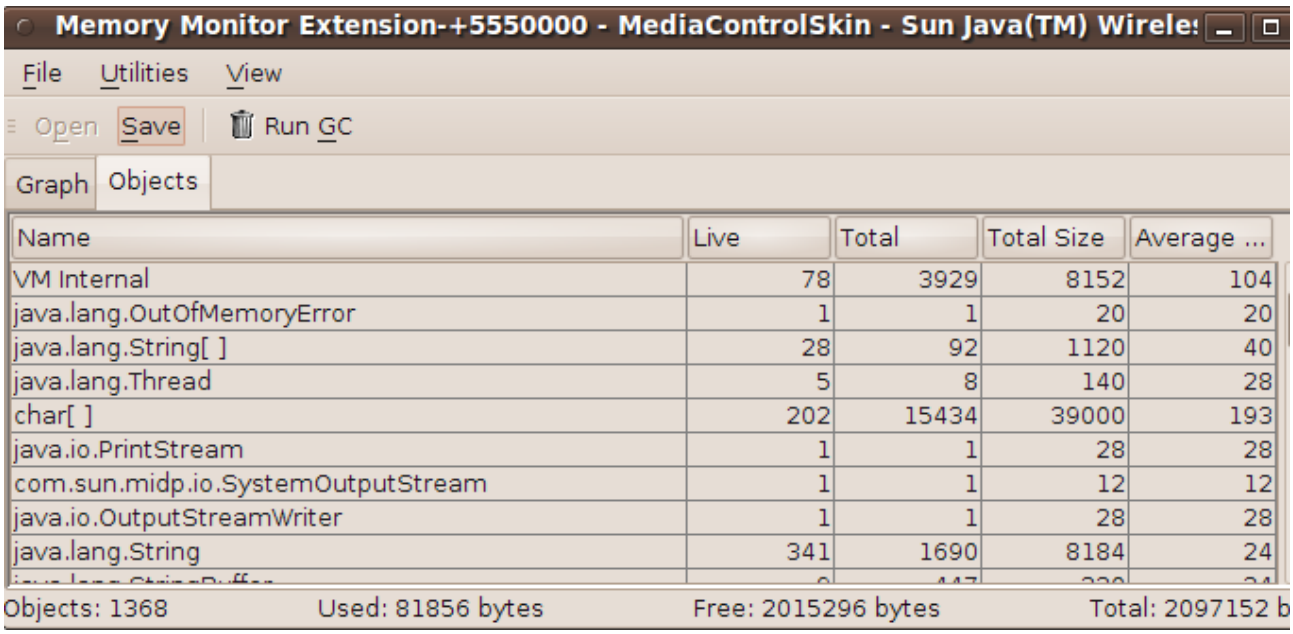
A leírásokban leginkább a személyes tapasztalataink alapján írunk (amelyek totálisan szubjektív vélemények), természetesen tényekkel kiegészítve.

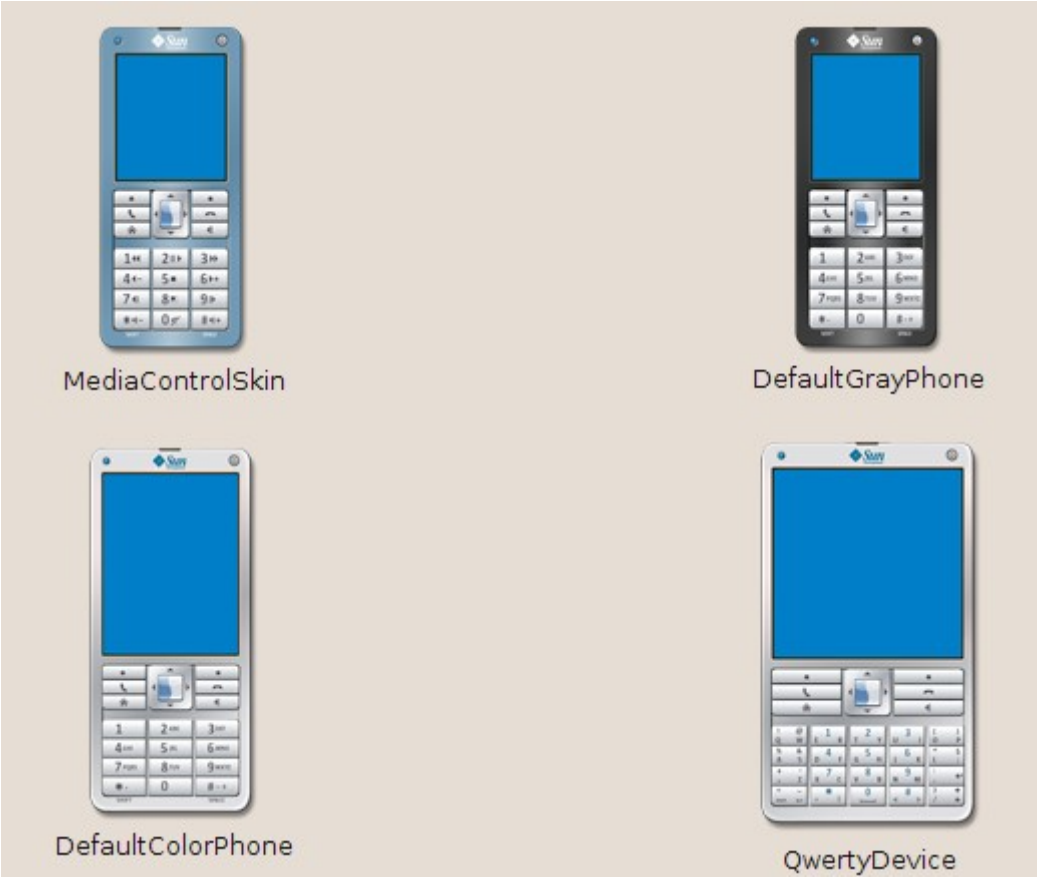
Eclipse

Ingyenes szoftver, mint a nevéből is kiderül, a készítők szerettek volna egy, a NetBeans–től különböző IDE-t, és mivel a Sun cég gyártja a Netbeans-t, így a totálisan ellenkező filozófiára utalás tükrében lett Eclipse a neve (Sun : nap, Eclipse: napfogyatkozás). Ez az IDE is nagyon sok mindent támogat, sőt merem feltételezni több mindent mint a NetBeans, több plugin is készül hozzá. Ebben az IDE -ben található a mobilprogramozáshoz az Eclipse-ME-re keresztelt plugin. Érdemes még megemlítenünk a PDT nevű plugin-t mellyel nagyon nagy támogatást nyújt a PHP fejlesztéshez.

Netbeans

Netbeans 6.7-es verziószámnál tart jelenleg. Ezt az IDE - t a Sun cég fejleszti és ingyenes szoftver. Nagyon sok plugin-nel rendelkezik, gyakorlatilag a teljes Java eszközrendszert kihasználhatjuk. Természetesen található plugin az ME részhez is (Mobility-nek keresztelt plugin). Mi ezt fogjuk használni. Vannak az ME részhez fontos feature-ök, például memóriahasználat követése:különböző kinézetű emulátorokat használhatunk, annak tükrében, hogy milyen kijelzőjű telefonra fejlesztünk:





és még sorolhatnánk a sok a hasznos tulajdonságát.

Ki kell emelnünk egy nagyon fontos részét az IDE-knek, mégpedig a grafikus tervezői felületet. Gyakorlatilag grafikusan „összehúzogathatjuk” az általunk használni kívánt eszközöket (pl: egy form, és arra gombokat, beviteli mezőket, stb.), és az IDE automatikusan legenerálja a hozzá szükséges kódot, amivel annyi a dolgunk, hogy implementáljuk az üzleti logikát, majd futtatjuk, és az eredményt „azonnal” láthatjuk. Egyetlen hátránya van a kódgenerálásnak az, hogy nehezen szerkeszthető a programkód, vagy egyáltalán nem szerkeszthető. (A mi példánkban, az általunk használt eszközrendszerekkel kapcsolatban nem kell tartanunk ettől, ugyanis minden kódot mi fogunk írni.) Kivételt képeznek, a Dia2Code, illetve a Netbeans: Code Generate-ekkel generáltatott kódok. (Osztály diagramok UML-ben történő tervezése, majd kód generálása. Ez nem egyezik meg az imént említett résszel.) Ebből fakadóan a tervezéshez is használjuk ezeket az eszközöket, IDE-ket. Szerepeltetni fogunk néhány diagramot, amiket Netbeans-el, illetve JDeveloper-el készítettünk, terveztünk. Természetesen van még egy tervező eszköz, amit használunk. Ez a Dia névre keresztelt alkalmazás. Ezzel készült diagram lesz a szerverhez illetve az adatbázishoz kapcsolódó dia mindegyike, JDeveloper-el készült diagramok lesznek a Use Case-ek, illetve az Activity

diagram, valamint Netbeans-el készülnek el a mobil – kliens osztály (Class) diagrammjai.

Elméleti háttér

Szerver – Kliens architektúra

Tervezés lévén, architektúrális mintákat használunk, ha tudunk természetesen. Fontos kérdéseket kell feltennünk önmagunknak a tervezés folyamán:

- Milyen specifikációs eszközt használunk: Amit mi már eldöntöttünk: UML.
- Mire optimalizálom a rendszert: Erről később.
- Tudunk–e újrafelhasználni valamilyen mintát: természetesen igen: ez lesz maga a Szerver – Kliens architektúrális minta (Osztott architektúra, MVC).

Erről az architektúráról bővebben:

Éles határ van a szerver és a kliens között. A szerver szolgáltatásokat nyújt, a kliens pedig szolgáltatásokat kér. Általában a szerver egy példányban létezik, míg kliensből bármennyi lehet. Nagy előnye további kliensek hozzáfűzése, a rendszer fejlesztése is könnyűvé válik. A kommunikáció XML alapokon történik, amiről már volt szó, hogy miért is hasznos („...az XML pedig a platformfüggetlen adatábrázolást és adatcserét teszi lehetővé...”).

Két rétegű kliens – szerver architektúráról beszélhetünk: vékony – kliens, architektúra vagy vastag – kliens architektúra.

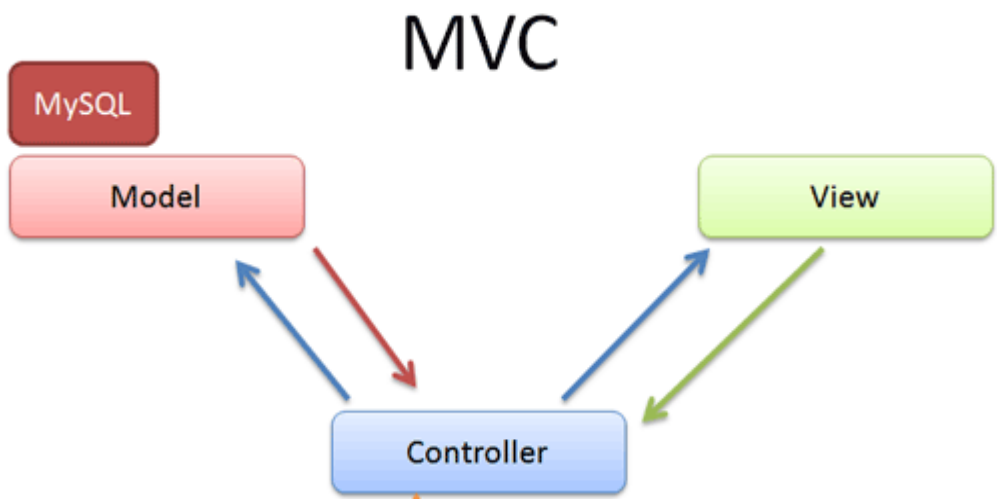
Vékony – kliens: a kliens a megjelenítésért felelős, míg minden másért (adatkezelés, alkalmazás feldolgozás) a szerver. A kliensek általában terminálok, PC-k. Hátránya, hogy a rendszer nem skálázható, és nagy adatforgalom a jellemző.

Vastag – kliens: a szerver az adatkezelésért felelős, a többit a kliens végzi. Kisebb az adatforgalom, de a leterheltsége nagyobb a kliensnek, és nyilván kliens oldalról már szükséges a nagyobb számítási teljesítmény.

A két rétegű megoldás átvihető három rétegűre is. A háromrétegű architektúra

gyakorlatilag három, egyenként önálló funkcionalitással bíró egységből áll. Az első réteg az adatbázis. A második réteg az alkalmazás maga, az üzleti logika, minden nagyobb erőforrást igénylő folyamat itt zajlik**,** a felhasználó csak az üzleti logikán keresztül képes elérni az adatbázist, ami egy nagyfokú biztonságot is jelent. A harmadik réteg a felhasználói felület, amely ha lehet, akkor platformfüggetlen, általában valamilyen Web technológia. A rétegekből egyenként akár több is lehet, a lényeg, hogy a feladatok jól el vannak különítve. Az egyes rétegek megvalósításánál szinte alig kell figyelembe venni a másik két réteg szoftvereinek típusát, mindig az adott feladatra legmegfelelőbb szoftvert választhatjuk. Nagyon jól a skálázható, a biztonság is nagyobb. Mi is ez alapján készítjük az alkalmazást (MVC).

Három rétegű architektúra (MVC – Model – View – Controller (MNV – Modell – Nézet - Vezérlő)):



Az MVC egy tervezési minta. Három fajta objektumot tartalmaz. A Model az alkalmazás objektum, a View annak ábrázolása a képernyőn, a Controller (Vezérlő) pedig annak módját határozza meg, ahogyan a felhasználói felület válaszol a felhasználói bemenetre. Pontosan az a célja, hogy elkülönítse az üzleti modell funkcionalitását a prezentációtól és a vezérlőtől, amelyek használják ezt a funkcionalitást, így növelve az újrahasznosíthatóságot, és a rugalmasságot.

Adatbázis tervezés

Mint minden háromrétegű alkalmazás esetén így a web-szerver mobil-kliens alkalmazások esetén is mind tervezéskor, mind implementáláskor érdemes az adatbázissal kezdeni. Mielőtt azonban a példa alkalmazásunk tervéről, illetve annak implementációjáról beszélnénk, érdemes szót ejteni az adatbázis tervezés mikéntjéről is.

Adatbázis tervezéskor minden alkalmazás esetében, így itt is, szeretünk egyedekről (entitásokról) beszélni, ugyanis ez nagyobb szabadságot ad a tervezés folyamán. (Szerencsére, mint azt látni fogjuk, a használni kívánt eszközrendszer kifejezetten támogatja ezt a fajta megközelítést, így valójában ettől az absztrakt felfogástól csak nagyon ritkán kell majd elszakadnunk.)

Viszont mielőtt bármilyen tervezésbe kezdenénk, el kell döntenünk, hogy milyen adatbázis vezérlőt választunk. Ennek a döntésnek a meghozatalakor sok kérdés felmerülhet, viszont az egyértelműen látszik, hogy jelenleg egyelőre a hatékony adatbázis kezelők körében csak relációs modellt valló adatbázis kezelők vannak. Tehát a modell jelenleg nem lehet kérdés. Az hogy ezen belül milyen adatbázis kezelőt választunk leginkább attól függ, hogy milyen rendszert tervezünk. Alapvetően az adatbázisok szempontjából kétféle rendszer létezik, az egyik az úgynevezett OLTP (OnLine Transaction Processing) illetve OLAP (OnLine Analytical Processing), ezek leginkább az adatbázison végzett műveletek szempontjából különbözőek.

Az OLTP rendszerek azok a mondhatni "hagyományos" adatbázis felhasználású rendszerek, azaz sok adat amelyekhez nincsenek igazán bonyolult lekérdezések, viszont esetenként rengeteg adatot dolgoznak fel, és az ilyen feldolgozásokból nagyon sok van nagyon rövid idő alatt. Majdnem minden esetben ezek a valódi real-time rendszerek. Ilyenek például a bankok esetében az ATM rendszerekkel végzett műveletek. Ezekből nagyon sok van, és nagyon rövid idő alatt nagyon sok kis műveletet kell végrehajtani. Ezeknél a rendszereknél a kritikus szempont a gyorsaság. De lényeges szempont lehet a biztonság, a megbízhatóság, és a nagy rendelkezésre állás. Az kézenfekvő, hogy ezek az igények nagyon ritkán találkoznak. A nagyobb adatbázisgyártók mint például az Oracle a hangsúlyt a biztonságra, a megbízhatóságra, az optimalizálhatóságra, és a rendelkezésre állásra helyezik, ami nem feltétlenül jelenti azt, hogy az Oracle adatbázisrendszerek

lassúak lennének, de a gyorsaság elég relatív fogalom, fontos kérdés, hogy mihez viszonyítunk. Az tény, hogy kis adatmennyiség esetén például az Oracle adatbázis kezelő lassabb, mint mondjuk a MySQL, vagy a PostgreSQL viszont minél több az adat, és minél komplexebb adatbázisokról van szó, annál gyorsabbnak tűnik az Oracle adatbázis kezelő a MySQL illetve PostgreSQL adatbázis kezelőkhöz képest. Ennek nem az az oka, hogy nagyobb mennyiségnél az Oracle adatbázis kezelője látványosan felgyorsulna, sokkal inkább az , hogy kiegyensúlyozott teljesítményt nyújt, aminél az adatok mennyisége nem igazán játszik szerepet. Ez más adatbázis kezelőknél (kivéve talán az IBM DB2-t) nincs így. Természetesen nem arról van szó, hogy mindegy lenne, hogy egy lekérdezés egy 20 elemet tartalmazó, vagy egy 20 millió elemet tartalmazó táblán fut, sokkal inkább arról van szó, hogy ugyan az a lekérdezés ugyanazon a táblán ugyanolyan sebességgel fut, ha az adatbázis 20MB méretű, mint amikor 2TB méretű. Tehát amikor az adatbázis kezelő rendszert választjuk meg, lényeges kérdés az adatok mennyisége, és azok biztonsága is. A webes környezetben általában az aránylag kis mennyiségű adatok a jellemzőek, amelyek általában nem államtitkok, tehát a biztonság bár lényeges, de korántsem annyira, hogy ilyen nézőpontok alapján kompromisszumot kelljen kötni a sebesség terén. Éppen ezért a webes környezetben elterjedtebb adatbázis kezelő rendszerek a kisebb, de kis adatok esetén fürgébb rendszerek, ezek közül is a legelterjedtebb a MySQL, ami mind sebességben, mind megbízhatóságban is kiemelkedő, tehát általában kisebb adatmennyiség esetén jobb választás. *Talán érdemes megjegyezni, hogy a sebességet említettük mint fontos szempont, ugyanakkor nyilvánvaló, hogy a mobilhálózaton folytatott kommunikáció miatt egyértelműen nem az adatbázis lesz a szűk-keresztmetszet, hanem sokkal inkább az adatok továbbítása, így tulajdonképpen a választásnál a sebesség nem is lenne annyira szempont. Viszont általában azért nem az Oracle adatbázis kezelőjét választják az ilyen fejlesztések során, mert túlságosan nagy, és azon túl, hogy az embernek "ágyúval verébre" érzése támad, fölöslegesen is foglal el túl sok helyet mind a memóriában, mind a háttértárolón.*

Nagyjából az adatbázis kezelő rendszer megválasztásának szempontjait fel is soroltuk az OLTP rendszerek esetén, viszont még nem esett szó az OLAP rendszerekről. Ezekről manapság elég sokat lehet hallani, egyre több gyártó megpróbál ilyen rendszerekhez eszközöket adni, megkönnyíteni az ilyen rendszerek fejlesztését. Ezek a rendszerek általában az úgynevezett BI (Business Intelligence) rendszerek. Ezekről sokat ezen dolgozat keretében nem érdemes szólni, ugyanis nyilvánvaló, hogy ilyen

rendszereket nem web-szerver környezetben, és főleg nem mobil-kliens környezetben szoktak fejleszteni. *Talán annyit azért mégis érdemes megemlíteni, hogy az adatbázis gyártók közül vannak akik külön BI rendszerekhez fejlesztenek adatbázisokat, viszont a kicsik közül alapvetően egyik sem foglalkozik kifejezetten ezzel a területtel, ennek leginkább az lehet az oka, hogy ezeket a rendszereket nagyvállalati felhasználásra szánják, és ott már amúgy is valamelyik nagy gyártó adatbázisrendszere fut. Talán érdekes lehet még, hogy az Oracle mint a legnagyobb adatbázis gyártó cég nem tervez külön BI (OLAP) adatbázis kezelő rendszert, sőt sokáig a már létező adatbázis kezelő rendszerét sem módosította ilyen irányban, mivel úgy tűnt az ilyen felhasználásra leginkább csak hangolni kell az adatbázisokat, és nincs szükség különleges fejlesztőeszközökre, vagy bármilyen belső átalakításra. Azonban úgy tűnik ezzel a filozófiával szakítani kívánnak a 11g megjelenésével, ugyanis egy sor olyan eszközzel bővítették az eszközkészletét az adatbázis kezelőnek, amelyek mind a BI rendszerek fejlesztését segítik elő. Ezek egy részét úgy helyezték el az adatbázis kezelőbe, hogy nem dokumentálták azokat, ez a módszer a kiforratlan eszközök tesztelésére, illetve finomítására nem ismeretlen az Oracle -nél, ebből is látszik, hogy elég komolyan gondolják az ilyen irányba történő elmozdulást.*

Miután mérlegeltük az adatbázisunk felhasználásának körét, és az rendszer működésére nézve kritikus szempontokat megvizsgáltuk, és eldöntöttük, hogy melyik adatbázis kezelő rendszert választjuk, még mindig előttünk áll az adatbázis tervezésének feladata, ehhez nem árt néhány alapszabályt belátnunk, mielőtt bármilyen tervet is készítenénk. Néhány szabály nem annyira általános az adatbázis tervezésnél a web-szerver mobil-kliens alkalmazások esetén, így most főként ezekről lesz szó. *Természetesen az adatbázis tervezés folyamatának többi része hasonlóan zajlik, mint bármilyen más alkalmazás esetén, mi csak azokról a módszerekről ejtünk szót amelyek főleg a mobil-kliens miatt lényegesek lehetnek.*

Az adatbázis tervezésekor először is azt kell látni, hogy milyen szereplői lesznek a rendszernek, és ezek mentén kell egyedeket képezni, majd meghatározni csoportokat (egyedtípusokat), amiket valamilyen közös jellemző alapján határozunk meg. Nagyon fontos tényező azt látni, hogy kik is fogják használni a rendszert, mert ezek a személyek egy-egy olyan csoportba tartoznak majd, amikhez külön-külön biztonsági kategória tartozik. Ennek lényegességét később látni fogjuk.

Az egyedtípusok meghatározása nem minden esetben egyértelmű, előfordulhat olyan eset, hogy egy-egy típushoz tartoznak olyan attribútumok, amelyek többértékűek lehetnek, ezeket természetesen meg kell szűrni, és valamilyen módon a többértékű attribútumok értékeinek kell találni valamilyen kategóriát, tehát ezeket is egyedtípusokká kell alakítani, és ekkor már a többértékűséget le lehet bontani, valamilyen egyedek között értelmezett kapcsolatra.

Ami kritikusan fontos tényező, hogy előre látni kell, hogy a mobil készüléken milyen egyedek fognak megjelenni, ugyanis ezeknek a típusait meg kell próbálni úgy meghatározni, hogy lehetőleg minél kevesebb legyen belőlük, minél kevesebb attribútummal, az attribútumokra nézve minél kevesebb információval, és ami nagyon lényeges: lehetőleg nulla egyedkapcsolati függőséggel. Ezekre azért van szükség, mert ezek az egyedek előreláthatólag a kommunikációban is részt fognak venni. Ebből kifolyólag minél kevesebb van belőlük, minél kevesebb attribútumuk van (vagy abból minél kevesebb vesz részt a kommunikációban, bár ez nem tartozik szorosan az adatbázis tervezéshez), illetőleg azok minél kisebb adatmennyiséget hordoznak, úgy ezzel csökkenthető a kommunikációban résztvevő adatmennyiség. Ez több szempontból is hasznos, ugyanis egyrészt a kommunikáció a mobil eszközöknél az egyik legfontosabb szűk-keresztmetszet, illetve a kommunikáció növelheti leginkább az alkalmazás költségeit. Hasonló okok befolyásolják azt is, hogy az egyedkapcsolatokat kerülni kell ezeknél az egyedtípusoknál, ugyanis ha egy egyed megjelenik a mobil oldalon is, és ahhoz több másik egyed is kapcsolódik, akkor azokat is kommunikálni kell a kliens felé. Ez nyilván problémás a fentebb említett okokból kifolyólag is, viszont problémás a kliens eszközön is, ugyanis ez problémát okoz az adatfeldolgozáskor is, illetve problémát jelent a megjelenítésnél is. (De ezekről bővebben később lesz szó)

Tehát az egyedtípusok megtervezésekor fontos figyelni arra, hogy mely egyedtípusok jelennek meg a kliens oldalon is, de ezután érdemes megvizsgálni, hogy egy-egy ilyen egyedtípusnak mely attribútumai lényegesek a mobil eszközön. Ugyanis ha ezeket előre látjuk, akkor kialakíthatunk olyan egyedkapcsolatokat, amelyek az adatbázis szempontjából fontosak, viszont garantáltan nem vesznek részt a kommunikációban, és a kliens oldalon csak közvetett módon érzékelhetőek.

Végül pedig minden egyedtípuson (főleg a kommunikációban is résztvevők esetén) szükséges az egyedet azonosító attribútumok körét szűkíteni, lehetőség szerint egy attribútumra érdemes leszűkíteni ezt a kört (bár ez nem feltétlenül szükséges, viszont hatékonyabbá tudja tenni az alkalmazást). Ennél a szűkítésnél nagyon fontos odafigyelni, hogy az egyedet azonosító attribútum, vagy attribútumok egyike se legyen összetett típus, illetve ne legyen egyedkapcsolatot leíró attribútum. (Bár mint az később látszani fog ez tulajdonképpen ugyan azt jelenti.)

Lehetne beszélni még a relációs adatbázis tervezésről, illetve arról, hogy az egyedtípusokat, illetve egyedkapcsolatokat hogyan érdemes leképezni relációs adatbázisra, viszont mivel ezt az entitáskezelő eszköz meg fogja tenni helyettünk, így ezzel nem kell komolyabban foglalkozni.

Szerver oldali elméleti áttekintés

Ahhoz, hogy bármilyen szinten elkezdhessük tervezni, illetve implementálni az alkalmazásunkat a szerver oldalon, meg kell ismernünk mind a nyelvet, mind a felhasználni kívánt eszközrendszert amiben fejleszteni kívánunk. Először is a nyelv amiben fejleszteni kívánunk az a PHP, tehát néhány szót fogunk ejteni ennek szerkezetéről, különlegességeiről, és kicsit a hiányosságairól viszont mivel a dolgozat témája nem alapvetően a PHP nyelv bemutatása, így nagyon sok dolgot feltételezünk, hogy az olvasó is ismer a nyelvvel kapcsolatban. Az eszközrendszer, aminek segítségével fejleszteni fogjuk a példa alkalmazásunkat, az pedig egy saját fejlesztésű nyílt forráskódú API aminek a neve PEA (PHP Enterprise API) minthogy ennek jelenleg még épp fejlesztés alatt áll az első publikus verziója, így ennek bemutatására több hangsúlyt fogunk fektetni, mind ebben az áttekintésben, mind pedig a példa alkalmazás fejlesztését bemutató részben.

A legalapvetőbb tudnivalók a PHP-val kapcsolatban, először is, hogy egy objektum-orientált script programozási nyelv. Éppen ezért a benne végzett fejlesztések akkor hatékonyak, hogyha kihasználjuk a nyelv objektum-orientált mivoltát.

A PHP egy gyengén típusos nyelv, ami azt jelenti, hogy bár létezik benne a típus fogalom, nagyon is határozottan, de alapvetően a nyelv az implicit típuskonverziót vallja, éppen ezért a változónak nem kell, és nem is lehet típust definiálni. Tehát minden változó típusa az értékadás közben fog eldőlni. Ettől függetlenül a nyelv ismer olyan primitív típusokat, mint a fixpontos egész számokat ábrázoló típus (int, integer), lebegőpontos számokat ábrázoló (float), karaktertípus (char), karakter tömb típus (string), tömb típus, asszociatív tömb típus (array), dátum típus, dátum és idő típus (date, datetime), és természetesen a logikai típust (boolean), és ezek esetében van mód az explicit konverzióra, amelyre, bár az implicit konverzió általában sok problémát megold, néha szükség is van. Természetesen mint objektum orientált nyelv a primitív típusokra építkezve definiálhatóak benne osztályok, és természetesen ezek példányaiként objektumok. Az objektumok az implicit konverzióban kicsit kivételek, mert ha egy objektum nem alakítható string-gé, akkor az kivételt okozhat a felhasználásban.

Az implicit konverzió főleg a hasonlító operátoroknál okozhat problémát, tekintve,

hogy például a null érték, amelyet bármilyen változó felvehet szintén átmehet implicit konverzión, és a null értéknek van minden primitív típusra nézve egy megfeleltetése, ami nem minden esetben szerencsés. Például ha egy boolean értéket várunk egy hasonlító operátor valamelyik oldalán szereplő változóba, és az operátorral azt a változót hasonlítjuk, akkor ha a változó értéke null volt, akkor a hasonlításban mint hamis érték fog részt venni, az implicit konverzió miatt. Éppen ennek kiküszöbölésére léteznek a PHP-ban ekvivalencia hasonlító operátorok, amelyek konverzió nélkül vizsgálják az értékeket, és a változók típusát is figyelembe veszik a hasonlításkor.

Fontos tudni még a nyelvről, hogy nagyon magas szinten támogatja a reflexiót, azaz azt a folyamatot, aminek során az egyes objektumok, a saját, vagy környezetük futási paramétereit befolyásolhatják. Ennek legjobb példája, hogy PHP-ban egy foreach ciklussal bejárhatóak egy adott objektum látható attribútumai, anélkül, hogy ehhez bármilyen különleges reflection osztállyal kellene bánnunk, vagy például egy attribútumra hivatkozhatunk egy szabályos a nyelvben értelmezett string segítségével. (Félreértés ne essék, ez nem az implicit konverzió része, azaz nem arról van szó, hogy nyelvi elemek is részt vesznek az implicit konverzióban, ez pusztán csak a reflexió egy módja.)

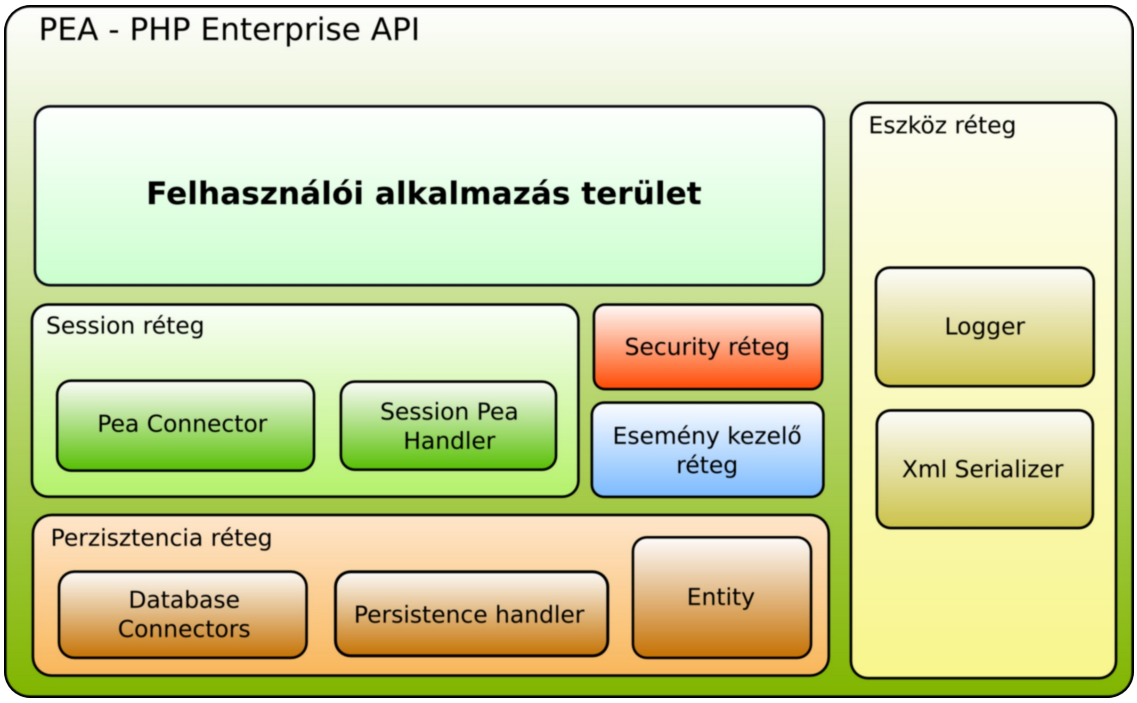
A nyelv furcsaságairól, szépségeiről, és hiányosságairól szintén lehetne egy külön szakdolgozatot írni főleg, hogy a PHP nyelvi változásai éppen a mostani verzióváltásokkal kezdődtek meg csak igazán, de sajnos ennek a dolgozatnak nem ez a fő témája, így most csak a példa alkalmazás szempontjából lényeges részekre fogunk kitérni, aminek fontos része a PEA eszközrendszer. Talán érdemes megmagyarázni, hogy miért is fejlesztettük ki ezt az eszközrendszert, ennek oka szorosan kapcsolódik a PHP felépítéséhez. A PHP bár nagyon kényelmes eszközrendszer, nem ad olyan szintű eszközrendszert, mint a Java EE, ami főleg az adatbázis vezérlése szempontjából, illetve a kommunikáció szempontjából fontos probléma. Egy a példa alkalmazáshoz hasonló alkalmazás fejlesztésekor ütköztünk abba a problémába, hogy bár a PHP eszközei nagyon kényelmesen használhatóak, de sajnos felépítésükből fakadóan a velük való fejlesztés elég erőteljesen probléma specifikus megoldásokat ad. Ez pedig probléma lehet a fejlesztés kibővítésekor, arról nem is beszélve, hogy az adatbázis vezérlés, illetve a kommunikáció olyan általános feladat, amelyet nem szabadna minden fejlesztéshez implementálni, ugyanis ez rengeteg időt vesz el a valódi üzleti logika implementálásától. Tehát a fejlesztés egy kényszerből indult, mégpedig, hogy az alap eszközrendszer nem

használható általános célokra, a lehetőséget pedig pont a PHP zseniális felépítése adta, azaz pont a magas szintű reflexió lehetősége. Ennek mentén lehetőségünk volt egy a Java EE környezetére hasonlító környezet kialakítására.

Ebben a környezetben létezik az átlagos szerveroldali fejlesztés két legfontosabb eszköze, az entitások, illetve a session objektumok. Alapvetően ezen eszközöket kell implementálni a fejlesztés folyamán, minden másról (perzisztencia kezelésről, adatbázis kialakításról, naplózásról, biztonságról, kommunikációról, session kezelésről, eseménykezelésről mind-mind a környezet gondoskodik)

Természetesen mivel ezen környezet fejlesztése még nemrégen kezdődött, így még sok hiányossága van, és előfordulhat, hogy egyes részek még hibásak, de ezektől eltekintve erre az eszközrendszerre építkezve lehet a legkényelmesebben PHP alapokon web-szerver mobil-kliens alkalmazásokat fejleszteni.

A PEA környezet megismeréséhez először is az alapvető struktúráját kell megérteni, amely a következőképp épül fel:



Természetesen ezen az ábrán csak a fontosabb elemek látszanak. Amit meg kell figyelni az ábrán, hogy ez a rétegződés az adatbázis felől az üzleti logikát megvalósító felhasználói alkalmazás felé halad. Továbbá maga az alkalmazás is részévé válik a rendszernek, és a működése csak a felhasználói alkalmazással értelmezhető, tehát nem egy függvényhalmazról hanem ténylegesen egy környezetről van szó.

Haladván az ábrán alulról felfele azaz az adatbázistól a felhasználói alkalmazásig fogjuk most bemutatni nem túl részletesen az egyes részeket. Tehát kezdjük is a Perzisztencia rétegbeli Entity (Entitás) ősosztállyal: A PEA-ban a perzisztencia fő eszközei az általános értelemben vett entitások, tehát amikor adatbázisbeli tárolásról beszélünk, akkor általában egyedek tárolásáról beszélünk. Az egyedek a PEA-ban az Entity ősosztály leszármaztatásával definiálhatóak. Alapvetően nem csak tárolt egyedek léteznek, minthogy megeshet, hogy egy-egy egyed csak valamilyen származtatott információt szolgáltat, amelyet nem szükséges tárolni. Az egyedek lehetnek tehát tároltak, és nem tároltak (származtatottak), és az egyedek soha nem ekvivalensek a példánnyal aminek az osztályában definiálva lettek. Az objektum tehát csak hordozza, illetve prezentálja az entitást, de valójában az entitás csak egy fogalom, amelyet az adatbázis oldalon általában

egy táblabeli sor, az alkalmazás oldalon pedig egy objektum reprezentál. Az objektum, mivel nem ekvivalens az entitással, így magának az objektumnak lehetnek saját attribútumai, és független funkcionalitása is. A tárolt entitások esetén az entitás attribútumait elő kell írni, és minden attribútumhoz kell legyen megfelelő attribútuma az objektumnak is, ami reprezentálja az entitást. Egy tárolt entitásnak lehetnek nem csak tárolt attribútumai, tehát a származtatott értékek nem csak a származtatott egyedek esetében léteznek. A származtatott értékek feloldásáért pedig maga a perzisztencia kezelő felel. (Tehát bizonyos értelemben ezek leginkább a kényelmesebb használatot szolgálják.) A tárolt entitásokhoz mindig kell rendelni egy adatbázis táblát (nem szükséges, hogy a tábla létezzen, minden entitásnak van olyan örökölt funkcionalitása, amivel létre tudja hozni a táblát, amiben az adatait fogja tárolni.), illetve a tárolt attribútumokat is jelezni kell, illetve azok típusát is elő kell írni. Az egyes egyedek hivatkozhatnak egymásra, illetve a tárolt vagy származtatott attribútumaikban tárolhatnak más egyedeket, ezeknek a kezelése a rendszer feladata, viszont a kötelező kapcsolatokat elő kell írni minden egyed definiálásakor, ugyanis a rendszer csak ezeket a kapcsolatokat tudja megvédeni az esetleges nem megfelelő felbontásoktól. Például ha egy egyed egy attribútumában tárol egy másik egyedet, és ez a kapcsolat a működés szempontjából nélkülözhetetlen, akkor elő kell írni ennek a kapcsolatnak az ellenőrzését, és ilyen esetben a hivatkozott elem törlését meg tudja akadályozni a rendszer, ezzel megóvva a kapcsolatot a két egyed között.

Az entitások definiálása a perzisztencia kezelés első lépcsőfoka, de az egyedeket kezelni is kell, bár maguk az egyedek sok funkcionalitást örökölnek az Entity osztályból, és ezzel részben meg is oldódik a kezelésük, viszont szükséges, hogy az egyes egyedeket ki is tudjuk választani az adatbázisból. Erre pedig a perzisztencia kezelő ad lehetőséget, amely áll elsősorban a PersistenceHandler osztályból, továbbá olyan kiegészítő osztályokból, mint például a Where osztály, ami a select utasítások where záradékának magas szintű megfelelője. Maga a PersistenceHandler osztály egy kibővített singleton osztály, ami annyit jelent, hogy az alkalmazás egyes funkcióinak megfelelően a különböző feladatokhoz külön-külön példányok létezhetnek belőle, de minden feladathoz maximum egy. A különböző feladatokat pedig úgy kell érteni, hogy egy alkalmazás esetén előfordulhat, hogy más-más sémákhoz, vagy eleve más-más adatbázisokhoz szeretnénk csatlakozni egy időben. Erre ad módot a kibővített singleton osztály. Minthogy egy PersistenceHandler osztályhoz csak egy adatbázis kapcsolat tartozhat, így

szükségszerűen több kapcsolathoz több perzisztencia kezelő kell, viszont emellett szükség van a singleton osztályok nyújtotta kényelemre (abból a szempontból, hogy az osztály ismeretével mindig elérhető a példány), így ez az osztály olyan formában kibővített singleton, hogy az osztály ismerete önmagában nem elég a megfelelő példány kiválasztásához, hanem egy paraméter ismeretére is szükség van, ami magát a példányt fogja azonosítani, viszont a példányok tárolásáról nem kell külön gondoskodni. A perzisztencia kezelőnek a feladata a hozzá tartozó kapcsolat menedzselése, entitások lekérdezése, (természetesen nem csak entitásokat lehet lekérdezni a segítségével) és a tranzakciók kezelése. Továbbá a perzisztencia kezelő az, amely tartalmaz mindig egy egyedkezelőt, amely egy valamilyen specifikus adatbázis-kezelő nyelvén definiált kezelő, amely az egyes magas szintű műveleteket végzi el adatbázis specifikusan. Éppen ez a részlet teszi teljesen függetlenné az adatbázistól a perzisztencia kezelést, és ez teszi lehetővé, hogy minden az egyedeken végzett műveletet magas szintű műveletként tudjunk értelmezni. Maga az egyedkezelő (EntityHandler) egy interfész, amelyet mindig az adott adatbázis specifikus kezelőnek kell implementálnia. Erről a kezelőről több szót nem fogunk ejteni, mert bár ez a rész talán a működés szempontjából az egyik legfontosabb része a perzisztencia kezelésnek, viszont mivel ezt az eszközt a perzisztencia kezelő, és maga az Entity ősosztály használja, és magas szintű műveletein keresztül elrejti előlünk, így nem szükséges ezzel foglalkoznunk. A perzisztencia kezelő tartalmazza továbbá az adatbázis kapcsolat menedzseléséért felelős DataBaseConnector-t is, ami szintén egy interfész, amelyet a megfelelő adatbázis specifikus kapcsolókkal kell implementálni.

A perzisztencia réteg után a második legfontosabb réteg a session réteg, ennek segítségével oldható meg a kliens oldal kommunikációja a szerver oldallal. Ez általánosságban úgy történik, hogy az üzleti logikát a szerver oldalon olyan osztályokba implementáljuk, amelyeket a kliens oldal a session rétegen keresztül tud használni. Alapvetően az elképzelés az, hogy a kliens oldalon megjelennek ugyan ezen osztályok interfészei, és ezek szolgáltatják a szerver oldalon implementált logikát, ezzel egy simább átmenetet képezve a kliens és a szerver közti határ között. Ez PC-s környezetben általában elég egyszerűen is implementálható, viszont sajnos mobil környezetben ez rengeteg tárhelyet foglalna, és koránt sem lenne ennyire kényelmes eszköz, így a mobil kliensek esetében nem jelennek meg ezek az osztályok, viszont ugyan úgy lehet velük bánni, mintha valódi osztályokkal bánnánk, csak a folyamat, ahogy ezeket kezeljük egy kicsit más. A szerver oldalon alapvetően kétféle osztályt definiálhatunk, az úgynevezett

Stateless, illetve Stateful osztályokat. A Stateless osztályok tulajdonsága, hogy nem tárolnak állapotokat, éppen ezért nincs is szükség a példányai megtartására. (Ezt úgy kell érteni, hogy két szerveroldali hívás közt az állapota nem tárolódik, sőt maga a példány, amit használunk sem lesz ugyanaz a példány.) A Stateful osztályok ezzel szemben megtartják az állapotaikat, és így elméletileg a teljes folyamat (session) alatt ugyanazzal a példánnyal dolgozhatunk. (Természetesen ez nem így van a valóságban, de a működés szempontjából ez így tűnik.) A PEA-ban az imént említett két osztálytípust bármilyen osztályon előírhatjuk, az osztályoknak nem kell leszármazniuk semmilyen előre definiált ősosztályból, csak azt kell megjelölni az osztályokon, hogy milyen típusúak, illetve hogy mely metódusaikat szeretnénk ha elérnének a kliensek. Ennek módja pedig, hogy az osztályra egy sémát írunk elő, amelynek segítségével definiáljuk a metódusok, paraméterek, illetve a felhasznált osztályok, entitások XML-beli megjelenését, ezzel el is tudjuk különíteni a különböző kliensek által meghívható metódusokat, látható attribútumokat, és gyakorlatilag az egész szerveroldali alkalmazás láthatóságát, és felhasználásának módját befolyásolhatjuk. Alapvetően kétféle klienst szoktunk megkülönböztetni ilyen módon: a "desktop", azaz a PC klienst, illetve a "mobil" azaz mobil klienst, de természetesen nem szükséges ezen a szinten megállni, lehet további csoportosításokat is végezni, akár annak alapján, is, hogy egyes mobil eszközök milyen képességekkel bírnak, illetve, hogy a PC-s környezetben milyen jogosultságai vannak az egyes klienseknek. Felépítését tekintve a session réteg a következő két osztályra támaszkodik leginkább: PeaConnector - Ennek feladata a kommunikációs protokoll alapján értelmezni a kéréseket, és vezérelni a szerver oldali folyamatokat, de feladata csak a kommunikáció első rétegéig tart, tehát ez az osztály hasonlatos az EntityHandler-höz, olyan értelemben, hogy ez az osztály alacsony szinten kommunikál a kliens oldallal, de nem implementál semmilyen konkrét vezérlést, csak a bejövő illetve a kimenő kommunikációt vezérli; SessionPeaHandler - Ennek az osztálynak a feladata a magasabb szintű vezérlés megoldása, illetve a Stateful példányok állapotának megőrzése is. Továbbá ez az osztály felel a biztonságos metódushívásokért is, melyet közvetett módon a Security réteg segítségével old meg.

A Security réteg felel a Session réteg által használt osztályok biztonságos kezelésért. Ennek módja a következő: Az alkalmazásban a szerver oldalon definiálhatunk biztonsági csoportokat, ennek módja, hogy definiálunk olyan felhasználókat, amelyek valamilyen módon egy biztonsági csoportba tartoznak, ezzel implicit módon létrejön egy

biztonsági csoport is, amibe a felhasználó fog tartozni. A felhasználók alapvetően bármilyen osztály példányai lehetnek (a biztonsági csoportot természetesen az osztályon lehet előírni), de érthető okokból érdemes entitásokat használni a felhasználók reprezentálására. A Security rétegben az egyes felhasználók bejelentkezését, illetve kijelentkezését lehet jelezni (alapvetően nincs implementálva semmilyen bejelentkeztető, illetve kijelentkeztető algoritmus, mivel ezek különböző esetekben különböző felépítésűek, és különböző erősségűek szoktak lenni, éppen ezért ennek implementálásának szabadságát szerettük volna meghagyni). A bejelentkezett felhasználónak kötelezően valamilyen biztonsági csoportba kell tartoznia, és így a bejelentkezett felhasználó alapján ellenőrizhető a metódushívások jogosultsága. Az egyes meghívható metódusokon előírhatóak engedélyek, és azok megvonása is az egyes biztonsági csoportokba tartozó felhasználók számára.

Érdemes még megemlíteni az esemény kezelő réteget, amelynek segítségével lehetőség van eseményszórással vezérelni az alkalmazásunkat, ezt a PEA egy olyan része használja, amelyet ebben a dolgozatban helyszűkében nem fogunk használni/bemutatni, de természetesen ettől még az alkalmazásban felhasználható ennek a rétegnek a funkcionalitása is. (A példa alkalmazásunkban módot is fogunk találni ennek a rétegnek a használatára.) Az esemény kezelés módja a PEA-ban a következőképpen néz ki: Először is definiálni kell olyan eseményeket amelyekre az egyes osztályok illetve azok metódusai feliratkozhatnak, mint annak az eseménynek a kezelői. Az események definiálására két mód is létezik, az egyik, hogy leszármaztatjuk az eseményt, ez a megoldás a jobb - később látjuk majd miért - a másik megoldás, hogy magát az Event osztályt példányosítjuk, és használjuk mint esemény. Az eseményekre a metódusok tudnak feliratkozni egy Trigger annotációval, ahol előírhatják, hogy milyen típusú események érdeklik őket (ezért érdemesebb leszármaztatni az Event osztályt), definiálható, hogy az adott esemény küldője milyen típusú legyen, illetve még definiálható egy kulcs is, amely az esemény zárját nyitja, de erről majd később. Az események a kezelőjükhöz való eljuttatásáért az EventHandler nevű osztály felel, mégpedig közvetlenül az esemény feladásának helyén. Az eseményeket az EventHandler sendEvent metódusával lehet feladni, amelyet ez a metódus el is juttat az esemény megfelelő kezelőjéhez, amiből természetesen több is lehet, hiszen eseményszórás alapú az eseménykezelés. Viszont ha egy adott eseményt azt szeretnénk, hogy csak egy kezelő tudjon kezelni, akkor ahhoz alkalmazhatunk az eseményen egy zárat, amelyet az imént

említett Trigger-beli kulccsal lehet feloldani, illetve, ha ez valamilyen okból kifolyólag nem alkalmazható (például dinamikus a kulcs előállítása), akkor az adott kezelő osztályában elhelyezett megfelelő metódus is feloldhatja az esemény zárát. Ezzel a megoldással kiküszöbölhető, hogy a nem megfelelő kezelőhöz jusson el egy olyan esemény, amire esetleg nem annak a kezelőnek kéne reagálnia. Természetesen ez a védelem fordítva is fennáll, azaz ha egy kezelő csak egyetlen eseményhez készült, akkor csak annak az eseménynek a zárát feloldva fog csak futni, tehát ha létrejön egy amúgy minden más attribútumaiban egyező esemény, amelynek nincs zára, akkor arra természetesen nem fog futni ez a specifikus kezelő.

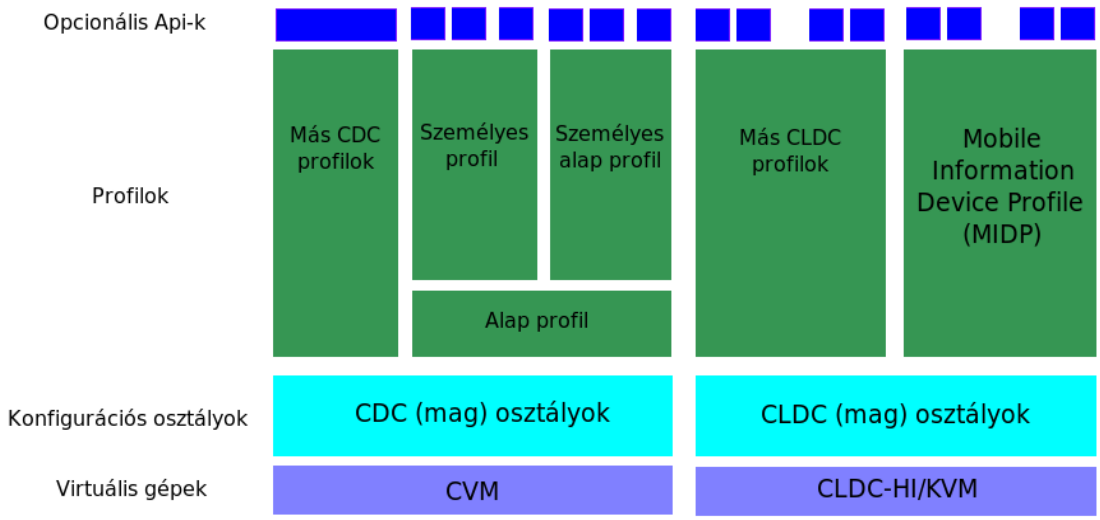
Az ábrán az eszköz rétegben két fontosabb elem lett kiemelve, az egyik a naplózás megoldásában nyújt segítséget (Logger) ennek a működése nagy vonalakban a következő: A Logger nevű osztály egy kibővített singleton osztály, ami azért lényeges, mert a különböző naplózásokat különböző naplóvezérlőkkel (LogHandler) lehet elvégezni. A naplóvezérlőnek egy nagyon egyszerű interfészt kell implementálnia, és ezen keresztül kapja meg a naplóbejegyzéseket, melyeket valamilyen módon tárol. A Logger osztályban található négy napló típus, amely az egyes bejegyzések típusát definiálja, ezek a következők: INFO, ERROR, WARNING, DEBUG; ezeken felül természetesen definiálhatóak mások is. (A definiálás implicit, tehát nem szükséges új konstansokat definiálni, ezek csak azért vannak az osztályban, hogy ne lehessen eltéveszteni az alap naplótípusokat.); Az eszköz rétegben a másik fontos eszköz amit kiemeltünk az az XmlSerializer amely egy osztály. Ezt az osztályt gyakorlatilag a PEA összes rétege használja valamilyen szinten, ugyanis a kommunikációhoz minden tekintetben XML formátumot használunk. Ennek megfelelően ez az osztály felel az összes alacsony szintű XML feldolgozási műveletért mindkét irányban, azaz az XML-re, illetve XML-ről való átalakításokért. Bár ez az osztály egy nagyon fontos szerepet tölt be a PEA-ban magát az osztályt nincs értelme bemutatni, mert bár lehetőség lenne rá, hogy a felhasználói alkalmazás közvetlenül használja, de erre nem lesz szükség, ugyanis a PEA összes rétegében felépülő elemek úgy lettek kialakítva, hogy ennek az osztálynak a használatát átvegye és elrejtse a felhasználói alkalmazás elől.

A fentiekben elég absztrakt szinten ismertettük a PEA főbb elemeit (illetve azokat, amelyeket használni is fogunk a példa alkalmazásunkban), ennél természetesen jóval részletesebb ismertetésre lesz szükség, hogy a példa alkalmazás forráskódját értelmezni

tudjuk, de erre pont ezen alkalmazás bemutatása folytán fogunk sort keríteni. Viszont van még egy nagyon fontos elem, amiről nem ejtettünk szót, vagy csak nagyon felületesen említettük. Ahhoz hogy a PEA-ban a definíciók kényelmesen használhatóak legyenek szükség volt egy deklaratív eszközre, amely a PHP-nak jelenleg natív módon nem része, ez pedig az úgynevezett annotációk. A Java-ban az 1.5-óta megjelent annotációkra nagy igény támadt a PHP-ben is, de natív módon nem került be a nyelvbe, viszont a PHP adta nagyfokú reflexió segítségével ez az eszközrendszer is könnyűszerrel implementálható. Ennek módja nem más, mint hogy az egyes osztályok attribútumain, metódusain, sőt magukon az osztályokon elhelyezett dokumentációs megjegyzésekhez hozzá lehet férni a futó alkalmazásban. Innen nyilvánvalóan egyenes út vezet bármilyen deklaratív programozási eszköz definiálásához. Viszont nyilvánvaló az is, hogy ha már létezik egy előre kidolgozott módszer ahogyan ilyen deklaratív leírásokat lehet megadni, akkor azt érdemesebb átvenni, így a Java-ban megismert annotációkra nagyban hasonlító (néhol azt kibővítő) annotációkkal dolgozhatunk a PHP-ban egy nagyon egyszerű eszközrendszer segítségével. Ez az eszközrendszer része a PEA-nak is, hiszen részben erre építkezve épül fel maga a teljes API is. *Meg kell jegyeznünk, hogy az annotáció kezelő rendszer nem saját művünk, az eredeti verziót Jan "johno" Suchal készítette, és publikálta az alábbi címen: http://code.google.com/p/addendum/ Addendum PHP Reflection Annotations néven LGPL (GNU - Lesser General Public License) licenc alatt.*

Kliens – oldali elméleti áttekintés

Már említettük, hogy a kliens programozása Java Micro Edition-ben fog történni. Ahhoz, hogy elkezdhessünk programkódot írni a nyelven, meg kell ismerni egy kicsit közelebbről: A Java ME, mint architektúra:



A Java ME mint architektúra definiálja a standard *konfigurációs beállításokat*, *profilokat*, és az *opcionális API*-kat (Application Programming Interface - Felhasználói Program Interfész). Ezen elemekből épül fel a teljes Java – futtatókörnyezet (JRE, Java Runtime Environment).

Egy Java ME - Konfiguráció egy virtuális gépet, valamint meghatározott könyvtárhalmazt jelent. Az azonos konfigurációk biztosítják az egy kategóriába tartozó eszközök számára az olyan alapfunkcionalitásokat, mint a hálózati kapcsolat és a memóriakezelés. Jelenleg két Java ME konfiguráció létezik: a CLDC (Connected Limited Device Configuration) , valamint a CDC (Connected Device Configuration).

A CDC konfiguráció két fő szempont alapján tervezték, az egyik a JAVA SE -vel való kompatibilitás, a másik olyan készülékek támogatása amelyeknek korlátozott erőforrásokkal rendelkeznek, és így kapunk egy hozzávetőleg 2Mbyte RAM-mal és 2Mbyte ROM-mal gazdálkodó flexibilis Java-futtatókörnyezetet. Támogatja a teljes Java virtuálisgép-specifikációt (szálkezelést, biztonsági szolgáltatásokat, lebegőpontos

számítást, stb...).

A CLDC a mobiltelefonok szigorú memóriakorlátaihoz igazítva sokkal jobban korlátozott konfiguráció. A korlátozások kiterjednek a virtuális gépre és az osztálykönyvtárakra is. Nem támogatott a lebegőpontos számítás, natív kód, szálkezeléssel kapcsolatban a szálcsoportok, daemon-szálak nem támogatottak (végrehajtási szálak természetesen vannak). Sok esetben eltérnek az itt található osztálykönyvtárak az SE osztálykönyvtáraitól. Memóriakorlát 125-256Kbyte. (Nagyobb tudású, nagyobb memóriával rendelkező készülékek számára már a kibővített CLDC 1.1 - es specifikációja használható, ebben már például támogatott a lebegőpontos számítás is). Mindezek mellett a mi applikációnknak megfelelő az 1.0 – ás (JSR – 139) is.

Komplett Java futtatókörnyezet biztosításához egy adott kategóriájú eszközön, a konfigurációt magasabb szintű API-val vagy profillal kell kiegészíteni. Ezek határozzák meg az alkalmazás életciklus-modelljét, a user interfészt, illetve a specifikus, az eszközre jellemző tulajdonságok használatát.

Alap profil (Foundation profile, FP): A CDC profilok réteges szerkezetűek, így a profilok az alkalmazások igényei szerint használhatók. Ez a CDC legalacsonyabb szintű profilja, szolgáltatásait mélyen beágyazott, felhasználói felület nélküli alkalmazásokhoz nyújtja.

A CDC és a Személyes profil (Personal profile, PP) együttesével teljes GUI vagy internet alkalmazások fejleszthetőek. Teljes Java AWT - tel (Abstract Window Toolkit) rendelkezik, és desktop környezetre tervezett web-alkalmazások futtatása is a szolgáltatásai közé tartozik. (PP helyettesíti a PersonalJava™ technológiát, és a PersonalJava alkalmazások egy az egyben átvihetők a J2ME platformra.)

A személyes alap profil (Personal Basic Profile, PBP) a PP egy részhalmaza amely környezetet biztosít olyan alkalmazásokhoz, amelyek hálózati kapcsolattal rendelkező eszközökön futnak. Alapszintű grafikus környezettel és specializált grafikus eszköztárral rendelkezik. (Általában a CDC és FP felett mind a PP, mind a PBP megtalálható.)

A MIDP (Mobile Information Device Profile) és a CLDC együtt kihasználja a hordozhatóeszközök lehetőségeit, illetve minimalizálja a memóriahasználatot. E kettő olyan dinamikus és biztonságos platformot definiál, amely hálózati funkciókat képes ellátni, magas szintű grafikával rendelkezik, és a telefon billentyűzetének totális kihasználásával biztosítja a navigációt és az inputot (ez utóbbi a MIDP – hez tartozik). MIDP 1.0 – át sok mobiltelefon támogatja, ám elég szegényes az eszközrendszere, valamint amiben mi a gui

(Fire 1.2 LGPL) – t programozzuk, az szintén megköveteli a 2.0 – ás verziójú MIDP–et, így erről beszélünk bővebben. A MIDP 2.0 (JSR – 118) -nak számos hardver és szoftver követelménye van.

Hardver Követelmények:

- kijelző:

  96x54 felbontás, 1-bites színmélység, közel négyzet alakú pixel
- beviteli eszköz:

  egykezes billentyűzet vagy kétkezes billentyűzet vagy érintőképernyő
- memória:

  MIDP-nek 256Kbyte „nem felejtő” memória, 8Kbyte memória az alkalmazásnak, 128Kb Heap
- hálózat:

  kétirányú rádiós csatorna, korlátozott sávszélesség

Szoftver Követelmények:

- Kernel

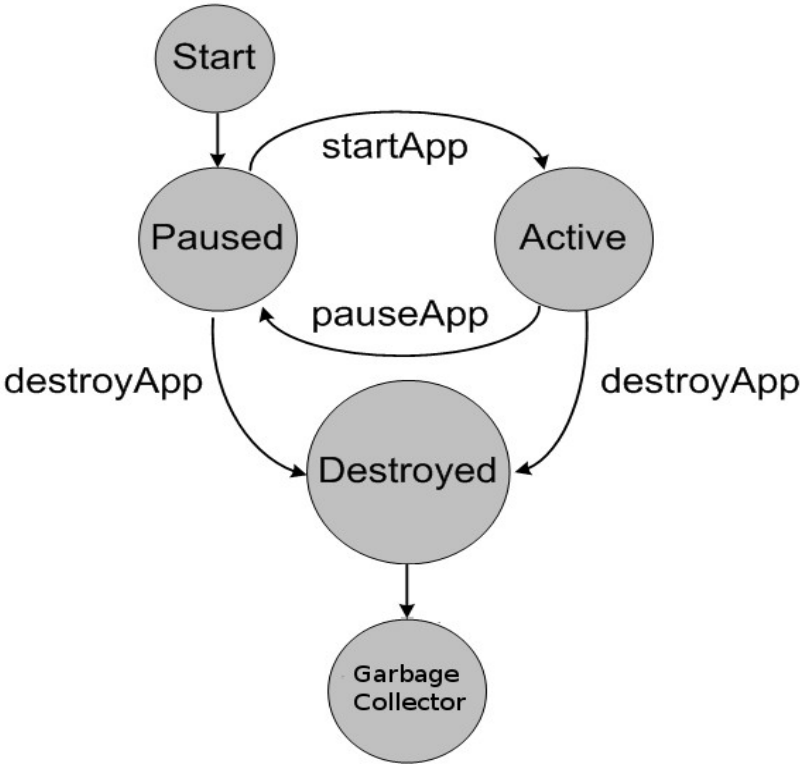
  Alkalmas legalább egy virtuális gép futtatására, valamint kezeli a hardvert

  (megszakítások, kivételek, minimális ütemezés)
- Időkezelés
- Az alkalmazás életciklusának kezelése.
- Minimális bitmap megjelenítési képesség a grafikus kijelzőn
- Olvasási és írási hozzáférés az eszköz rádiós hálózati kapcsolatán keresztül
- A készülék szoftvere biztosítsa a „nem felejtő” memóriából való olvasást illetve írást

Opcionális API–k további kiegészítések a piaci igényeknek megfelelően. Ezek standard API–k a létező és fejlődő technológiákhoz (Bluetooth, Web-szolgáltatások, multimédia...). Moduláris felépítése miatt a gyártók azt a komponenst illeszthetik az eszközhöz, amelyik szükséges. A CLDC és a MIDP fölött helyezkednek el ezek az Opcionális API-k.

MIDP-környezetben a futó alkalmazásokat MIDlet-eknek nevezzük, amelyeket egy alkalmazásmenedzser kezel (Application Managemant Software, AMS). A MIDleteknek saját életciklusa van. Az életciklus modell 3 állapotból áll: Active (Aktív, felhasználói képernyőkön keresztül kommunikáló MIDlet), Paused (felfüggesztett, nem használt de aktiválható – egy újonnan példányosított MIDlet mindig ilyen állapotban van,)

és Destroyed (halott, a MIDlet befejezte működését). A lenti ábra eléggé szemlélteti a folyamatot. A nyilak mentén szereplő szavak maguk a metódusok, amelyek meghívásával váltakoznak az alkalmazás életciklusai. Tehát ezeket a metódus - specifikációkat kell implementálnunk a programozás során.



Még muszáj említést tennünk a fájlrendszerről, a MIDlet–ek információinak tárolási módjáról. Ezt a JSR (Java Specification Request) - t, JSR – 75 (File Connection API) -nak nevezik. Fájlok létrehozása, olvasása, írása, törlése, könyvtárak kezelése az amihez eszközöket biztosít, viszont az alapértelmezett Java – ME specifikáció nem támogatja a fájlrendszert, fájlrendszer kezelését. Ennek oka, hogy a kezdetekben kevés tárhely révén nem merült fel a szükségessége. Így mi sem ezt az API–t fogjuk használni, hanem helyette az RMS (Record Management System) segítségével megvalósított adattárolást. Minden Java-ME alkalmazáshoz tartozik egy saját „adatbázis”, amelyet az alkalmazás MIDletei elérhetnek, valamint a tárolt adatokat olvashatják, módosíthatják és törölhetik is. Az RMS nem más mint egy rekord-orientált adatbázis-kezelő rendszer, amely perzisztensen tárolja az adatokat, amelyek byte formátumban vannak tárolva. Nem teszi lehetővé a Java objektumok, illetve típusok eltárolását, tehát konvertálnunk kell, de mivel erre vannak beépített eszközeink, így nem okoz nagy problémát. Az RMS használatához, szükséges a csomag importálása, amely megtalálható a javax.microedition.rms csomagban. Ez az adatbázis gyakorlatilag nem más mint egy RecordStore Objektum.

(Fontos megemlíteni, hogy egy alkalmazás MIDletei elérhetik a RecordStore–on keresztül egymás rekordjait, viszont az alkalmazások nem érhetik el a többi adatbázisban tárolt rekordokat.) A RecordStore osztály metódusait meghívva hozhatunk létre új rekordokat, törölhetünk, módosíthatunk. Létrehozásnál fontos, hogy a rekord neve maximum 32 unicode kódolású karakter lehet, illetve case–sensitive módon történik, tehát van különbség a kis- és a nagy betű között. Minden egyes rekordnak egyedi azonosítója van, amennyiben kitöröljük az adott azonosítójú rekordot, egy másik rekord akkor sem kaphatja meg az ő azonosítóját, magyarul nem kerül újrahasznosításra.

A kommunikációról is esett már szó, mivel ez egy elég fontos kérdése az alkalmazásunknak. XML alapú kommunikációt valósítunk meg, és ehhez szükségünk van egy elemzőre (Parser). Már esett szó a kXML–ről, említés szinten. Annyit szükséges tudni, amikor XML elemzőkről beszélünk, hogy ez az elemző igen csak jól van implementálva (Szintén szubjektív véleményről van szó.). A helyhiány miatt, ami vonatkozik mind az alkalmazás helyigényére, mind a szakdolgozat terjedelmességére, most sem fogunk többet beszélni róla. Ehelyett megírásra került egy saját kis elemző, ami nagyon problémaspecifikusan van implementálva, ennek nagy előnye a helytakarékosság. Mobil programozása során ez az egyik legfontosabb tényező, sok mindent meghatároz. Ennek kifejtéséről bővebben a gyakorlati részben.

A grafikus felülethez a FIRE (Flexible Interface Rendering Engine) -t használjuk, ami a GUI megjelenítéshez szükséges könyvtárakat jelenti. Komponens alapú grafikus eszközöket szolgáltat, amelyekkel sokkal gyorsabban, szebb kinézetet „varázsolhatunk” az alkalmazásunknak, mint a beépített eszközrendszerrel.

Fontosabb tulajdonságai:

- UI Funkcionalitása teljes mértékben MIDP alapú, tehát amely telefon támogatja a MIDP 2.0 – át, azon működni fog, illetve CLDC 1.0–ás profil tartozik még a követelményei közé.
- Themable UI, ami annyit jelent, hogy maga a kinézet beállítható egy png fájlal.
- Animált komponensek is implementálásra kerültek.
- Beviteli eszközökhöz interfész került implementálása.
- Felugró ablakok, menük, teljes képernyős nézet támogatása.
- Úgynevezett FTicker–ek használata, ami ugyanúgy működnek mint a J2ME Ticker–jei.

- Külön eszközt biztosít az I/O–nak.

Fontosabb osztályokról:

- FireScreen osztály:

  A FireScreen a magja az egész Fire engine–nek. Ez az egyetlen megjelenítő osztály, tehát az egyetlen GUI osztály. Ez egy singleton osztály, ami annyit jelent, hogy privát konstruktora van, és csak egy példány létezhet belőle a program futása során. Rakhatunk rá például Paneleket, PopupPaneleket egyszóval komponenseket. Egy 50x17–es méretű png fájl segítségével állíthatjuk be a téma (Theme) paramétereket. Ami annyit jelent, hogy pár ilyen png fájl megszerkesztésével könnyedén változtathatjuk az alkalmazás kinézetét. Néhány pixel megfelelőjét leírjuk, a többit az olvasó kíváncsiságára bízzuk (amit a Fire 1.2 dokumentációs leírásában megtalálhat). Ezt a png–t úgy használjuk, hogy (0,0) – (49,14) pixelek határozzák meg a logót, ami alapértelmezetten a jobb felső sarokba kerül. (16,0) – (16,15)–ig egyéb grafikus eszközök színeit állíthatjuk be: (16,0) a háttérszínt jelenti, (16,1) Label színe , (16,2) scrollbar, vagyis a csúszka színe. A megjelenítésen kívül, még ez az osztály felel a billentyűzet megfelelő beállításáért, ami még nem teljesen tökéletes, ami azt jelenti, hogy vannak néha kivételek, de elmondható, hogy nagy általánosságban azért működnek.
- Komponens Osztályok:

  Gyakorlatilag ezek az alapjai a GUI–nak, a komponens alapúságról beszélünk. Az itt megemlített osztályok mind – mind komponensek, kiterjesztik a *Component* osztályt. Ezek a komponensek eléggé hasonlítanak az SE–beli komponensekre. Volt olyan osztály amelybe bele kellett írnunk, kibővítenünk, hogy az általunk kitűzött célokra is használhassuk. Ezek az osztályok lesznek a *ListBox*, amelyek a listák megjelenítéséért felelősek, szoros kapcsolatban áll a *ListElement* osztállyal, ami a „grafikus lista” elemeit reprezentálja. Továbbá a *RowTextBox* osztály is átírásra került. Ez az osztály kiterjeszti a TextBox osztályt, ami egy beviteli box–ot jelent.

  A legfontosabb komponens a fontosak közül, az maga a *Panel* osztály. A FireScreen–en, az egyetlen megjelenítőn, mindig valamilyen Panel osztály egy példányát fogjuk megjeleníteni, amelyen elhelyezésre kerülnek a további

komponensek. A Panel osztálynak van egy esztétikailag szép tulajdonsága. Ha egy másik megjelenítésre szánt panelt jelenítünk meg, akkor animáltan fog előtérbe helyeződni az új panel, ami vagy jobbról, vagy balról fog becsúszni. Ehhez létezik egy metódus, amivel beállíthatjuk. A *Row* osztály felelős bizonyos elemek horizontális megjelenítéséért. Ezek az elemek lehetnek képek, szövegek, vagy akár mindkettő. Vannak eszközök az elhelyezésre, tördelésre, színbeállításra (a már említett png segítségével). Használható két más komponens közötti látható, vagy éppen láthatatlan elválasztóként is. Nagy előnye, hogy nem csak megjelenítésért felelős, hanem összekapcsolható egy Command–al, sőt, beállíthatunk akár saját CommandListener–t is és így működhet egyfajta grafikus gombként is, amelyre fókuszálva és klikkelve eseményeket indíthatunk el, vagy éppen állíthatunk le. (A Command osztály tulajdonságai közé tartozik, hogy megadhatjuk, hogy melyik billentyű legyen rá hatással, gyakorlatilag a CommandListener mire figyeljen, így beállíthatjuk akármelyik billentyűre.) Használható input bevitelre is, ezt a TextBox, illetve RowTextBox felhasználásával, (itt is felhívnánk a figyelmet, hogy miért is jó a komponens alapúság.) bizonyos metódusok meghívásával, és az ezek által kiváltott beállításokkal használható. Ezért is változtattuk meg a RowTextBox működését, mert bevitelre is fogjuk majd használni, és a funkcionalitásán kellett módosítanunk.

- FireIO osztály:

  Ez az osztály felel a már korábban említett RecordStore – ok kezeléséért. Annak ellenére, hogy csak az alapvető dolgok kerültek implementálásra, ez a rész nem került átírásra, totálisan fölhasználásra került, mivel éppen ennyi elegendő számunkra is, de ennél többre is használható: például képes letölteni képeket a kapott http címről, és azokat eltárolni egy RecordStore állományban, amit természetesen később felhasználhatunk, például a getLocalImage() metódussal.
- Lang osztály:

  Ez az osztály a nemzetköziesítésért felelős. Alapértelmezett nyelv az „EN”. Ezt mi nem használjuk, de használata egyszerű (Java-doc).
- FString osztály:

  Gyakorlatilag a komponenseken való szöveg megjelenítésére való. Megadhatjuk a Font–ot, illetve a szélességét a szövegnek, és ezek tükrében fog megjelenni.

Ezt az osztály csak annyiban írtuk át, hogy egyszerűen többsorosan is képes legyen a megjelenítésre, amit egy metódus meghívásával érhetünk el.

Röviden megnézzük, hogyan is áll össze ez az egész. Először is: beállítjuk a Theme–t, betöltjük a megfelelő png fájlt, fájlokat. Első ízben lesz egy FireScreen példányunk, az egyetlen FireScreen példány. A megfelelő metódusok meghívásával beállítjuk a kinézetet, megadjuk őt, mint alapértelmezett megjelenítőt. Megadjuk a megjeleníteni kívánt Panel–t, aminek célszerű Singleton–nak lennie, de ez szintén a panel felhasználásától függ. A panel példányon szerepelhet bármilyen komponens ami implementálásra került, de könnyen írhatunk mi is egyet. Az adott Panelhez adhatunk gombokat, ezekhez CommandListenert, és mindehhez funkcionalitást. Részletesebben a gyakorlati részben.

Gyakorlati háttér

A példa feladat bővebb leírása.

A bevezetőben leírt folyamatoknak mindnek menedzselhetőnek kell lennie a mobil kliensről, tehát listázni kell tudnia a még nem teljesített (és teljesítés alatt nem álló megrendeléseket), ezek közül le kell tudnia foglalni egyet, melyet a futár épp teljesíteni fog (ezt tehát a többi futár már nem foglalhatja le). Majd a szállítás végeztével jelentenie kell tudnia, hogy a szállítás teljesítve lett. Továbbá, ha voltak visszaküldött DVD–k, akkor azokat is jelentenie kell tudnia, melyek ekkor visszakerülnek az újra kölcsönözhetőek listájára. A rendszernek ezen kívül listáznia kell tudnia a futár felé, hogy mely DVD-ket hívta vissza valamelyik adminisztrátor. Ezeket a DVD-ket a rendszer automatikusan a visszahívott DVD-k listájára rakja. Majd a futár ezt a listát tudja megtekinteni, és ezek közül is tud választani, hogy melyiket próbálja meg visszavinni raktárkészletre. Tehát a futárnak a folyamatok elején lehetősége van eldönteni, hogy egy új szállítást intéz, vagy egy lejárt kölcsönzés kölcsönzési tételeit próbálja meg visszavinni raktárra. Mindkét esetben lehetősége van a megrendelő adatai alapján telefonon értesíteni a megrendelőt az érkezéséről, illetve informálódnia arról, hogy a megrendelő az adott időpontban tudja-e fogadni a futárt.

A rendszer szempontjából fontos szereplők az **adminisztrátor** jogokkal rendelkező felhasználók, akiknek a feladata menedzselni a DVD–k életciklusát. Tehát az adminisztrátoroknak lehetőségük van új DVD–t regisztrálni a rendszerbe, joguk van továbbá törölni egyes DVD-ket, illetve módosítani azoknak adatait. Lehetőségük van továbbá listázni a megrendelőket, illetve törölni egyes megrendelőket a rendszerből. Ezen kívül lehetőségük van listázni az egyes megrendeléseket, és törölni a teljesítés alatt nem álló megrendeléseket. (Tehát a teljesítetlenek, illetve a teljesítettek közül bármelyiket, viszont olyat nem amit egy futár éppen teljesít.) Továbbá jogukban áll a még le nem járt kölcsönzési idővel rendelkező kölcsönzött DVD-k visszahívására.

Általános követelmények:

- A fejlesztés folyamán előálló szoftver, forráskód, illetve dokumentációnak szabadon elérhetőnek kell lennie. (Azaz a szoftvernek nyílt forráskódúnak, és szabadon terjeszthetőnek kell lennie.)
- A rendszernek feladata folyamatosan tájékoztatást adnia a felhasználóknak az egyes folyamatokról, illetve hiba esetén kötelessége a hibáról és annak okáról tájékoztatni a felhasználókat.

Rendszerkövetelmények:

- Szerver operációs rendszer: Linux/Unix/Windows
- Mobil operációs rendszer: CLDC 1.0 és MIDP 2.0 kompatibilis környezetet biztosító operációs rendszer.
- Adatbázis: MySQL 5.0 vagy újabb.
- Célhardver: A szoftver igényeihez fog igazodni, tervezéskor nem kell figyelembe venni.
- Mobil kapcsolati hálózat: minimum: GPRS (56-114 kbit/s)
- Várható felhasználók száma: maximum 1000 fő.

Tervezés:

Természetesen, mint minden alkalmazás esetében, így a példa alkalmazásunk esetében is az első lépés a tervezés, amelyet azért tartunk fontosabbnak ismertetni mint magát az alkalmazás teljes forráskódját, mert szemben a forráskóddal az általánosabb célú tervekből az olvasó számára sokkal világosabb lesz miről is szól maga az alkalmazás. Természetesen fogunk bemutatni forráskód részleteket melyeket magyarázni is fogunk, de ezekkel csak példa jelleggel teljesen kiragadott módon egy-egy érdekesebb funkciót, vagy működési módot szeretnénk bemutatni.

Magát a tervezést mint már említettük UML tervvel fogjuk megvalósítani, a következő lépésekben:

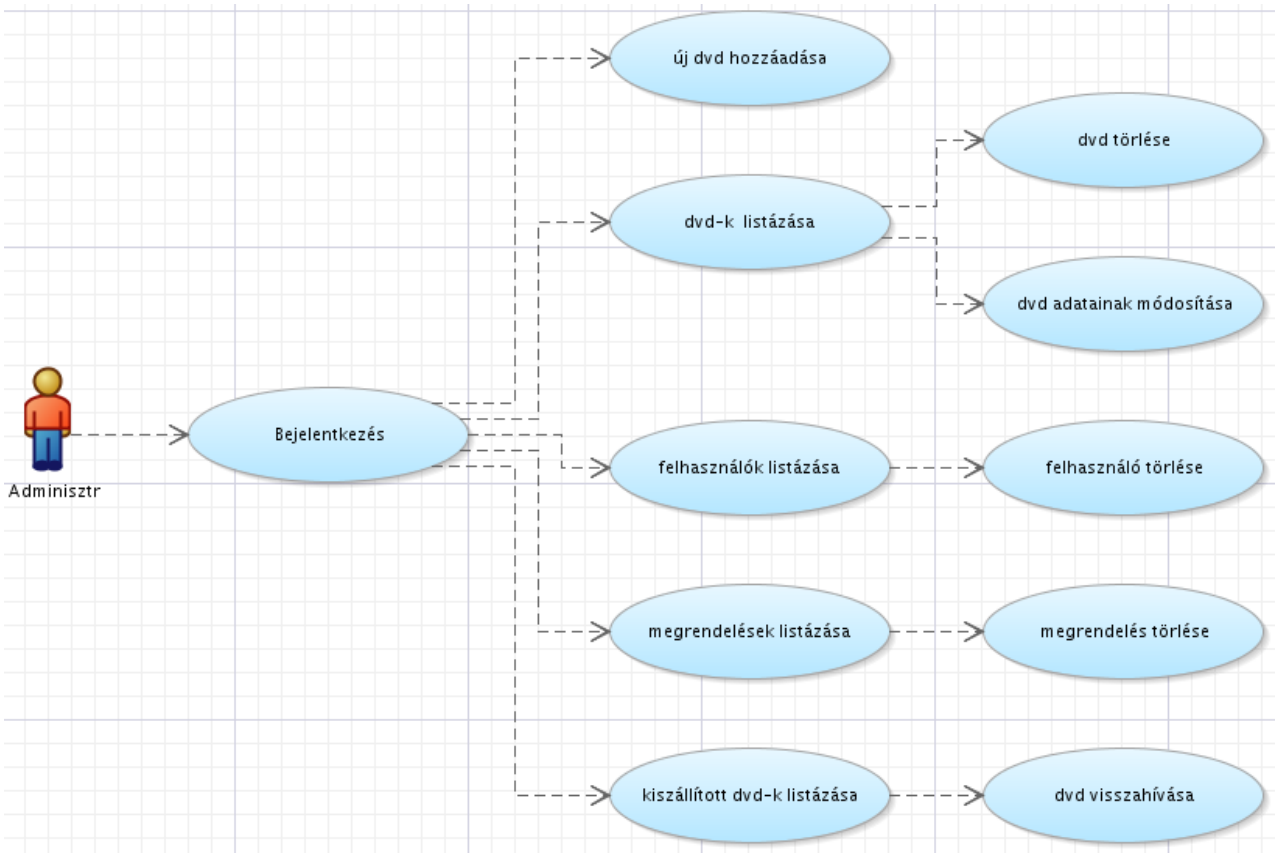
- Use-case diagram: A rendszer funkcióinak komplett leírására szolgál, azaz megmondja, hogy konkrétan milyen funkcionalitással kell bírnia, és mindezt úgy, ahogyan az egy külső szemlélő szemszögéből látszik. Ezen keresztül fogjuk demonstrálni az alkalmazások elvárt működését, azaz nagyjából azt, hogy a kész alkalmazásban melyik felhasználótípusnak milyen lehetőségei adódnak az alkalmazás használata közben. Pár szóban az elemeiről:
  - **Actor** – szerepkört definiál; a rendszer határain kívül eső szerepeket Actor-oknak nevezzük.
  - **Használati eset (Use Case)** – elvárt viselkedési minták(események sorozatát reprezentálják); minden mód, ahogyan egy Actor használhatja a rendszert (ezek a use case–k), minden használati esethez tartozik egy szöveges leírás: forgatókönyv. A bejelentkezés az Adminisztrátorhoz tartozó diagramon egy használati eset.
  - **Kapcsolat** – az Actorok, és a használati eset között asszociációs kapcsolat (irányított: kezdeményezési irányt mutató,irányítatlan) lehetséges, de csak ezek között (két Actor vagy két Use Case között nem).
- Mobil kliens Activity diagramja: A rendszeren belüli tevékenységek folyamatát reprezentáljuk. Általában üzleti folyamatok leírására szolgálnak ezek a diagramok, amelyek megmutatják, hogy az egyes tevékenységek egymáshoz képest mikor és milyen feltételekkel hajtódnak végre.
- Egyed kapcsolati diagram: Ezen keresztül fogjuk bemutatni, hogy a perzisztencia rétegben milyen egyedek jelennek meg, azoknak milyen tulajdonságaik vannak,

illetve, hogy azok hogyan kapcsolódnak egymáshoz. (Megjegyzés: itt nem a hagyományos ER diagramot fogjuk használni, hanem az osztálydiagram egy specifikusabb változatát, ami azért jobb, mert egyrészt áttekinthetőbb, mint az ER diagram, másrészt pedig a következő lépésből az osztálydiagram tervezésből "kipipáljuk" vele az egyedek tervezését.)

- Osztálydiagram: Az osztálydiagram egyszeresen összefüggő gráf, amelynek csomópontjai osztályokat, élei pedig relációkat fejeznek ki. Ezzel fogjuk bemutatni az alkalmazás végleges komplex belső szerkezetét, természetesen nem garantált, hogy a tervezési lépésben megtervezett osztályokon felül az implementáláskor nem készülnek plusz kiegészítő osztályok, illetve hogy az osztályok nem bővülnek plusz attribútumokkal, vagy metódusokkal, viszont az garantált, hogy a tervezés ezen fázisában az alkalmazás szempontjából fontosabb osztályok mindegyike megjelenik az osztálydiagramon. Pár szóban az elemeiről: Az osztály jele egy három részre osztott téglalap, ahol a felső részben az osztály neve, a középsőben az osztály attribútumai, az alsóban pedig az osztály metódusai szerepelnek. A statikus adattagokat vagy metódusokat aláhúzással jelöljük, az absztrakt osztály neve pedig dőlt betűs. Az attribútumok és a metódusok láthatóságainak jelölése a következőképp zajlik: a public (publikus) láthatóságot egy '+' jellel, a protected (védett) – et egy '#' jellel, a private (privát) – ot '-' jellel, és a package (csomag) láthatóságot pedig egy '~' jellel jelöljük. Az osztályok közötti kapcsolatok: association (asszociáció/társítás), dependency (függőség), generalization (általánosítás), aggregation (aggregáció/rész-egész kapcsolat), realization (megvalósítás).

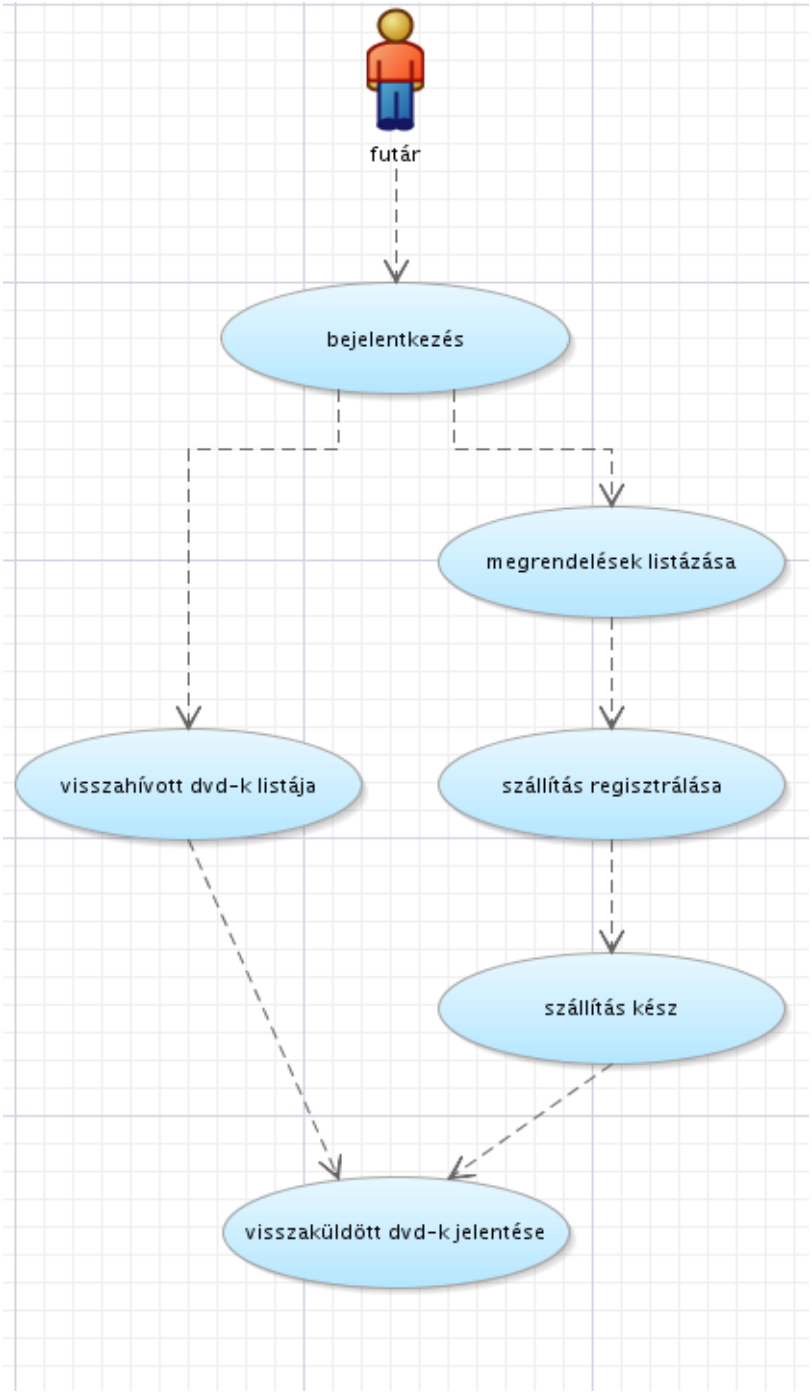
Use-case diagram:

Adminisztrátor:



Az adminisztrátor feladata az alkalmazás lényegét adó DVD-k illetve megrendelések menedzselése. Tehát a főbb funkciók amelyeket az adminisztrátor használni fog az ezekre korlátozódnak. Természetesen lehetőséget kell adni a felhasználók menedzselésére, viszont ez túl messzire vezetne, ezért csak a végfelhasználók törlésére adunk lehetőséget, hogy ezzel bemutassuk a lehetőséget, de nem szeretnénk, ha az alkalmazás nagyobb része a felhasználók kezeléséről szólna. Alapvetően tehát az adminisztrátor csak az egyedek kezelését végzi, viszont ehhez kap egy szebb grafikus felületet, amelyen keresztül kényelmesebben tudja elvégezni a módosításokat szemben az adatbázis közvetlen módosításával.

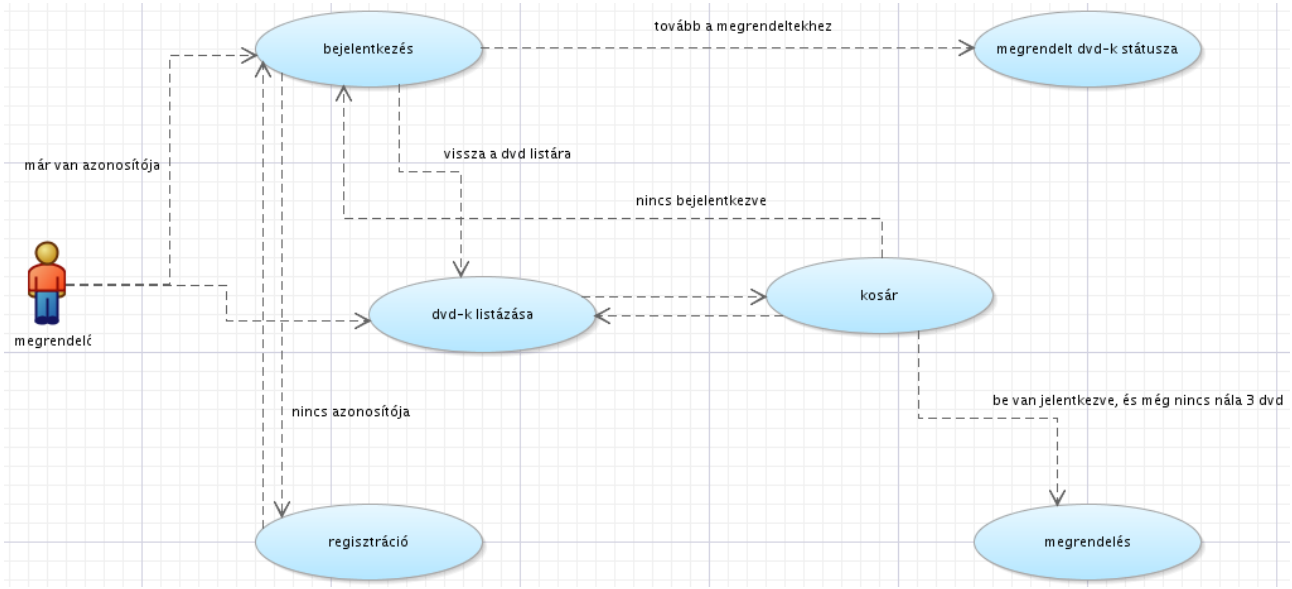
Futár:



A futár feladata a megrendelt DVD-k célba juttatása, illetve a visszaküldendő/visszahívott DVD-k raktárkészletre való visszaszállítása. Éppen ezért az futár esetében csak a mobil eszközről történő alkalmazás használat jöhet szóba. Tehát a futár lehetőségei eléggé korlátozottak, és irányítottak, ennek egyrészről az alkalmazás egyszerűsége az oka, másrészről viszont tudatos tervezés a mobil eszközön való navigálás nehézségei miatt. Alapvetően az egy fontos tényező már a tervezéskor, hogy a

mobil alkalmazás funkcionalitása ne legyen túl szerteágazó. Lehet bonyolult egy ilyen alkalmazás is, viszont azt mindig fontos szem előtt tartani, hogy valamilyen módon célzott legyen a mobil alkalmazás a felhasználásra, és minél kevesebb látható funkciót biztosítva, minél kevesebb hibázási lehetőséget is adva, egy kényelmes, és gyors felhasználású alkalmazást tervezzünk. Tehát a lényeg nem az alkalmazás funkcionalitásának szegényítése, hanem a hibázási lehetőségek, és a fölösleges átláthatatlanság csökkentése. Tehát az a mobil alkalmazás tekinthető jónak, amelynek a használata egyértelmű, és amiben az egyszerre választható lehetőségek száma kicsi, és a hibázás lehetősége elhanyagolható.

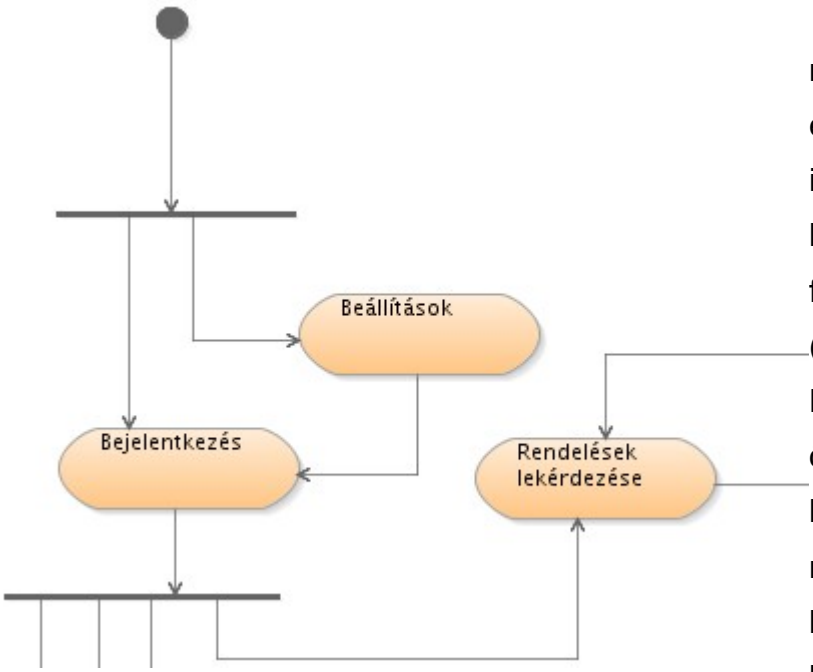
Megrendelő:



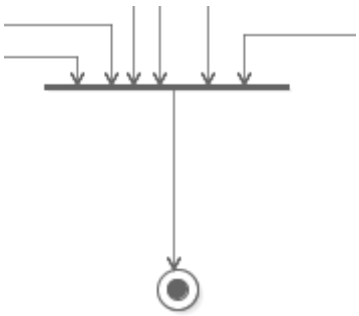
A megrendelő "feladata" az őt érdeklő DVD-kről egy megrendelési lista összeállítása, és annak megrendelése. Illetve lehetőséget biztosítunk még a már megrendelt DVD-k státuszának követésére, amely tulajdonképpen nem más, mint egy kontroll amely az alkalmazás valósidejűségét, illetve kapcsoltságát hivatott bemutatni. A megrendelési lista összeállítására a megrendelő egy "kosárba" rakhatja a kiválasztott elemeket, ezen keresztül szeretnénk demonstrálni az Session-ök működését. (Máshol is meg fognak ezek jelenni, viszont itt hangsúlyosabb lesz a szerepük.)

Kliens alkalmazás Activity diagramja:

Ez a diagram a futár feladatköréhez tartozik, és jól reprezentálja a majdani mobiltelefonján futó alkalmazás üzleti folyamatát. Már megbeszéltük, hogy mi a futár feladata, így most a diagram megértésére koncentrálnunk. A fontosabb részeket bemutatjuk:

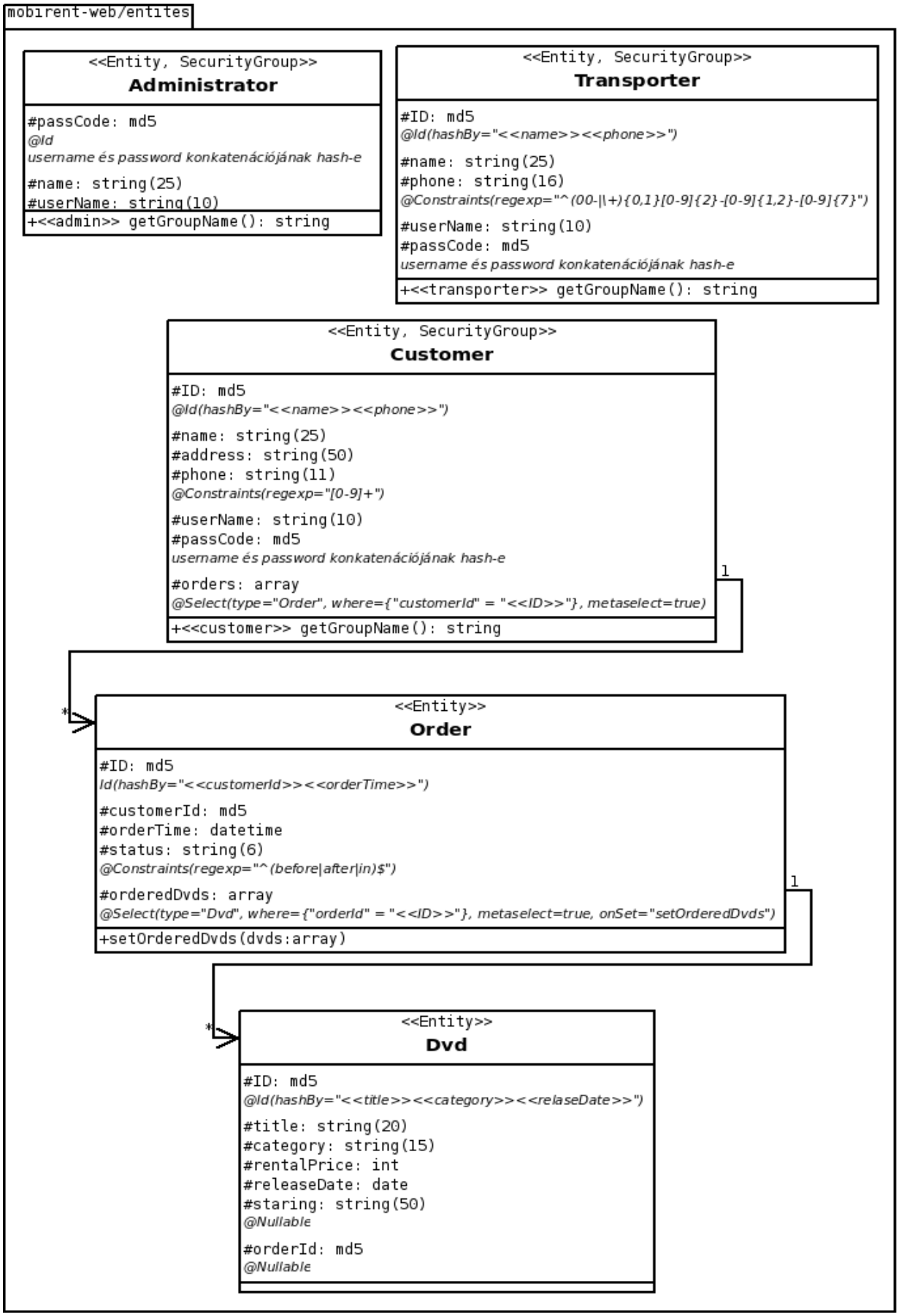


Amit legfölül látunk az reprezentálja az alkalmazás elindítását, majd több irányba indulhat el az alkalmazás használatával kapcsolatban a felhasználó, vagyis a futár (Bejelentkezés, Beállítások). Ezt a részt érdemes összevetni a Use-Case korábbi diagramjával, és megfigyelni hogy a mobil – kliens tükrében mit is jelent a két diagram (valamint a működés megértése szempontjából fontos). Az alkalmazás befejezését pedig jól láthatjuk a lenti képen (a nyilak száma azt mutatja, hogy az alkalmazásból hány helyről léphetünk ki):



Fontosnak éreztük elkészíteni ezt a diagram fajtát a futárhoz kapcsolódóan, a kliens könnyebb érthetősége végett.

Egyed kapcsolat diagram:



Mint az jól látható ez a diagram egyáltalán nem a megszokott ER diagram, ennek oka, hogy az implementáció folyamán az adatbázis objektumok "generálását" a PEA-ra bízzuk, és ez a diagramm tulajdonképpen a kibővített osztálydefiníciós diagramja az osztályként megjelenő egyedeknek. Tehát az alkalmazásunk első lépése a perzisztencia rétegbeli elemek tervezése egybe esik az egyed kapcsolat tervezésével. (Talán érdekes lehet, hogy a könnyebbséget nem csak az jelenti, hogy két tervezési lépés kapcsolódik össze, hanem azt is, hogy az így megtervezett osztályok generálhatóak is a megfelelő eszközökkel, például az általunk favorizált Dia2Code nevű programmal.)

Viszont ennek a diagramnak vannak olyan részei, amiket a hagyományos osztály diagramon nem szoktunk meg, mielőtt belemennénk az egyes osztályok elemzésébe előbb ezeket nem árt tisztázni. Az első ami észrevehető, hogy minden osztályon találhatóak sztereótípusok melyek részben azt is definiálják, hogy egyedekről van szó, ezt azért érdekes, mert minden osztály az Entity ősosztályból származik le, és ezt jelöljük vele. Továbbá vannak osztályok amelyek pontosításában található egy SecurityGroup nevű szereótípus is, ez tulajdonképpen a SecurityGroup interfész implementálását írja elő az osztályon. A jelentése pedig, hogy az ilyen osztályok példányai játszhatják a felhasználók szerepét - különböző biztonsági engedélyekkel, illetve megszorításokkal - az alkalmazásban. A SecurityGroup interfésznek egyetlen absztrakt metódusa van, a getGroupName nevű, amelynek egy stringet kell visszaadnia, és ez a visszaadott string lesz a biztonsági csoport neve, amelyikbe az adott egyedtípus által reprezentált személy tartozik. Ezeket pedig a metódus összes implementációs helyén sztereótípusokkal meg is adtuk.

A második érdekesség lehet, hogy minden attribútum protected láthatósággal szerepel. Hogy ez miért van így annak megértéséhez egy kicsit bele kell mennünk az Entity osztály működésébe, illetve a PHP elméletébe: Alapvetően a PHP-ban minden osztályra nézve elhelyezhető egy általános getter, illetve setter metódus, amelyek úgynevezett magic metódusok, tehát előre definiált a hívásuk, ezt a két konkrét metódust a PHP elemzője akkor használja, ha egy osztály olyan attribútumát akarjuk lekérdezni vagy beállítani, ami az adott helyről nem látható. Tehát tulajdonképpen ténylegesen automatikus és általános getter és setter metódusokról van szó. Ezekbe olyan komplex kezeléseket helyezhetünk amilyeneket csak akarunk. Tehát elsőre ennyit a PHP részről, az Entity oldalon pedig nyilván ezeket a metódusokat azért használjuk, hogy a szinkron az adatbázis objektumok és az osztályok példányai közt megmaradjon. (Nem egészen, illetve nem csak erről van szó, de első körben elég, ha ennyit elfogadunk.) Tehát előre definiáltan

vannak ezek az automatikus getterek illetve setterek az Entity osztályban, és az alosztályok attribútumait szinkronizálják. Mint azt már említettük ezek a metódusok csak a nem látható attribútumok beállításakor, illetve lekérdezésekor jönnek működésbe. Tehát logikus módon az alosztályokban egy olyan láthatóságra lenne szükség az attribútumokra, amely az osztályon kívül nem elérhető attribútumokat eredményez, és amire ezek a metódusok működni tudnak. Elsőre természetesen a private láthatóság tűnik tökéletesnek, de ha végig gondoljuk az objektum-orientált paradigma állításait, akkor könnyen rájöhetünk, hogy ez nem lesz jó nekünk. Illetve valamit még kell látni ami szintén PHP-val kapcsolatos dolog, mégpedig, hogy a java-hoz hasonlóan a PHP-ban is minden metódus dinamikus kötésű virtuális metódus, azaz ha meghívunk egy örökölt metódust, akkor az abban az osztályban fog futni, amelynek példányán hívtuk meg azt a metódust. Ez jelen esetben nekünk tökéletes, hiszen a getter setter metódusokat a leszármazott osztályok példányain már nem kell implementálni. Tehát a nyelv nem állná útját az ilyen szintű absztrakciónak, illetve a private láthatóság használatának. Viszont ezen a ponton egy kis összeférhetetlenségbe ütközünk az objektum-orientált paradigmával. Mégpedig azzal a részével, hogy a private tagok ugyan publikusak az osztályon belül, és privátok azon kívül, és ugye azt gondolhatnánk, hogy mivel minden metódus dinamikus, és virtuális, így azok számára az adott osztály példányában ezek az attribútumok publikusak lennének. Viszont ez a nézet téves, ugyanis van egy másik fontos kitétel is a paradigmában, mégpedig, hogy az örökölt metódusok számára csak azok az adattagok láthatóak, amelyek az öröklődésben láthatóak. És végül pedig van az öröklődésre vonatkozó szabályozás, ami azt mondja, hogy az öröklődésben csak a public és protected láthatóságú elemek láthatóak. Ebből pedig következik, hogy a mivel ezek a metódusok örököltek, így csak az öröklődésben látható elemekkel operálhatnak, azok pedig csak a protected illetve public elemek. Felmerülhet a kérdés, hogy akkor ez most tulajdonképpen fordított öröklődés-e. A válasz egyértelműen nem, hiszen ha belegondolunk, akkor az történik, hogy egy örökölt metódus fér hozzá olyan elemekhez amelyek az öröklődésben láthatóak, ez önmagában ugye azt a látszatot kelti, mintha fordított lenne az öröklődés, de mivel a metódusok mindegyike dinamikus kötésű, így nyilvánvaló, hogy ezek a metódusok az adott osztály példányain futnak, mégpedig úgy mintha az adott leszármazott osztályban lennének implementálva. Tehát ez a módszer sem technikailag, sem szemléletében nem sérti az objektum-orientált paradigmát.

A harmadik észrevehető eltérés a hagyományos osztály diagrammokkal szemben, hogy néhány attribútum után láthatóak annotációk, ezeket részletesebben

magyarázzuk majd az egyes osztályok magyarázatában. Ezek mint már említettük a forráskódban, a dokumentációs megjegyzésben fognak megjelenni, így itt is a megjegyzések részbe kerültek. Illetve még az lehet érdekes, hogy némelyik attribútum típusa után láthatunk egy-egy számot. Egész pontosan a string típusúak esetén vannak ilyen megkötések. Eleve a típusok megjelenése furcsa lehet, hiszen végül PHP-ban lesznek ezek az osztályok implementálva, ahol köztudottan nem kell és nem is lehet definiálni nyelvi szinten az attribútumok típusát. Nos ezek a típusdefiníciók nem is az osztályokra vonatkoznak, hanem az entitásokra. Mint azt már említettük a PEA-ban az entitások nem egyenértékűek azzal az osztállyal, ami definiálja őket. Tehát itt pontosan erről az elkülönülésről van szó, az entitások számára definiálni kell a típust, és a string-ek esetében definiálni lehet azok maximális hosszát is, ami az adatbázisba képzéskor meg is jelenik a táblák definíciójában.

Tehát mint említettük az a célravezető, ha az egyedeken egyesével végighaladva magyarázzuk az üzleti logikát, illetve az annotációk, és kapcsolatok szerepét az egyes egyedek esetében.

Administrator entitás: A rendszer adminisztrátorát (illetve adminisztrátorait, ha több is lenne) reprezentáló entitás, így alapvetően az adminisztrátorhoz mint személyhez kapcsolódó adatokat prezentálja. Tehát a fontosabb attribútumai az adminisztrátor neve, felhasználói neve, illetve a passCode nevű mező, amely a felhasználói nevéből illetve a jelszavából képzett md5 hash kód lesz. Erre azért van szükség, mert alap követelmény a rendszerek esetében a biztonság, és így nem tárolódik a felhasználó jelszava, így ennek biztonságos (titkosított) tárolásáról nem kell gondoskodni. Illetve, mivel más felhasználóknak nem lehet ugyan az a felhasználói nevük, és jelszavuk, így ez tökéletesen megfelel elsődleges kulcsnak. Ezen a mezőn látható egy Id nevű annotáció is, amelynek a funkciója azt megjelölni az Entity ősosztály felé, hogy az adott mező lesz az elsődleges kulcs. Elsődleges kulcsnak minden tárolt entitás esetében kell lennie, és csak és kizárólag az entitás szempontjából primitív típus lehet. Az md5 hash egy primitív típusnak tekinthető, és külön nevesítetten szerepel is az Entity osztály által elfogadott típusai közt. Az Id annotációnak lehet egy paramétere, mégpedig a hashBy paraméter, amelyet ebben a helyzetben nem használunk, ennek az lenne a funkciója, hogy ha az adott egyedtípus kulcsa md5 hash típusú, és a tárolt attribútumok közt vannak olyanok, melyek konkatenációjából képezhető egy egyedi azonosító, akkor az automatikus hashelést el tudja végezni helyettünk az Entity osztály. Ez jelen esetben azért nem használható, mert

mint említettük ennek az egyedtípusnak egyetlen egyedében sem tároljuk explicit módon a jelszót, így ez az automatikus hashelést kizárja.

Transporter entitás: A rendszerben a futárokat reprezentáló entitás, hasonlóan az adminisztrátorokhoz, itt is a személyekhez kapcsolódó fontosabb információkat tároljuk az egyes egyedekben. Ilyen a futár neve, telefonszáma, felhasználói neve, illetve passCode mezője. Jelen esetben is lehetne, hogy a passCode játssza az azonosító szerepét, viszont szerettük volna bemutatni a fentebb említett hashBy opciót. Így ennek az entitásnak az azonosítója az ID nevű mező lett, aminek a típusa md5 hash, és a hashBy attribútum az annotációban a ezt a stringet tartalmazza: "<<name>><<phone>>" ezeket a jelölőket mi meta-azonosítóknak szoktuk nevezni, mivel ezek visszamutatnak az osztály attribútumaira. Itt nagyon fontos kihangsúlyozni, hogy az osztály attribútumairól van szó, azaz nem az entitás attribútumairól. Így egy jóval bővebb lehetőségeket adó eszköz ez, mint azt majd későbbiekben látni is fogjuk. Jelen esetben ez annyira nem érdekes, mivel mind két mező tárolt mező. A phone azaz a telefon mezőnél található még egy eddig még nem ismert annotáció, ez a Constraints annotáció, ez mint a neve is mutatja, megszorításokat ír le az adott attribútumra, melynek teljesülését az attribútum beállításakor az Entity osztály felügyeli. Jelen esetben itt egy reguláris kifejezést adtunk meg mint megszorítást, ami az európai telefonszámoknak megfelelő módon kötőjelezett telefonszám leírást teszi kötelezővé a futárok telefonszámainál. Ez a mintaillesztés, ha nem teljesül a beállításkor, akkor minden esetben kivétel váltódik ki. A megszorítások annotációban, egyébiránt nem csak reguláris kifejezéseket fogalmazhatunk meg, hanem ha szám típusról van szó, akkor korlátozhatjuk a minimális, illetve a maximális értékét. (Ezzel oldható fel az is egyébként, hogy ha az adatbázisban egy ilyet típusra annak hosszát korlátozó előírás szerepel. Viszont ez ennél jóval nagyobb szabadságot biztosít, hiszen azzal a módszerrel nem lehet előírni egy fix tartományt.)

Customer entitás: A vásárlókat leíró entitás, ennek megfelelően a vásárlókhoz, illetve megrendelőkhöz kapcsolódó adatokat írja le, név, cím, telefonszám, felhasználói név, passCode, és ami érdekesebb lesz: a megrendelései. Itt már a telefonszámra nem kötöttünk ki túl bonyolult megkötéseket, nem kell se kötőjeleket se teljes telefonszámot megadni, csak számokat fogadunk el, és azokból a leghosszabb módon 11 fogadható el, ezt viszont nem kell a reguláris kifejezésben előírni, hiszen ezt a megszorítást már tartalmazza a típusdefiníció. Érdekesebb viszont ennél az egyedtípusnál, hogy van egy származtatott attribútuma, ami nem más mint az orders azaz a megrendelések attribútum, ennek ugyan meg van adva a típusa, hogy array, viszont ez lényegtelen, ugyanis ez az

attribútum nem egy tárolt attribútum. Ezen az attribútumon található egy Select nevű annotáció, ami azt mondja meg az Entity osztálynak, hogy ez az attribútum egy select típusú (azaz kapcsolatot leíró) attribútum, amit garantáltan nem kell tárolni. A Select annotációnak vannak főbb paraméterei, amelyeket meg kell adni, hogy a megfeleltetés működőképes legyen. Első sorban meg kell adni a típust, ami lehet egy másik hivatkozott entitás osztályának a neve, illetve lehet array is. Ez utóbbi esetről majd később, előbb nézzük azt az esetet, ami itt is fennáll, azaz hogy egy másik entitás osztályának nevére hivatkozunk. Ha csak ennyit adnánk meg a Select-ben akkor nyilván az összes egyedtípushoz tartozó egyedet megkapnánk. Ezért meg lehet adni a where záradékot. Ennek módja kétféle lehet: az egyik, hogy egy string-et adunk meg ami egy az egyben a leképzendő SELECT utasítás WHERE záradékába kerül. (Természetesen a meta-azonosítók nem, de egyelőre ettől tekintsünk el.) Illetve megadhatunk itt egy tömböt is (akárcsak ebben a konkrét példában), a tömb egy asszociatív tömb lesz, és a kulcsai jelentik majd az egyedtípushoz tartozó tábla oszlopait, illetve a kulcsokhoz tartozó értékek jelentik majd a kötelezően egyenlőséggel való megfeleltetéssel ellenőrizendő értéket. Természetesen megadható több oszlop = érték páros is egy-egy ilyen tömbben, ekkor ezek kapcsolata kötelezően AND kapcsolat lesz. A WHERE záradékban legyen az akár egyszerű string típusú, akár tömb típusú, az értékek lehetnek meta-azonosítók is, viszont mivel előfordulhat, hogy egy meta-azonosítóhoz hasonló string-et szeretnénk hasonlítani a where záradékban így alapértelmezésben ezek figyelése ki van kapcsolva, bekapcsolni pedig a metaselect nevű paraméterrel lehet, ami egy boolean. Mivel jelen esetben az Order-ben tárolt customerId mezőhöz szeretnénk hasonlítani a Customer ID mezőjét, így itt szükség van arra, hogy ennek értékére dinamikus módon hivatkozzunk, így ebben az annotációban a metaselect-et true-ra kell állítani. Mint fentebb említettük a Select annotáció type paramétere lehet array is ami azt jelenti, hogy a visszakapott elemek minden értelmezés nélküli tömbök lesznek, ez a módszer akkor lehet hasznos, ha a kiválasztásban szűrni szeretnénk egy-két oszlopra, és az adott helyzetben nem érdekel az entitás többi értéke. (Vagy nem is létezik az adott táblához entitás.) Ebben az esetben a minket érdeklő mezők neveit meg kell adnunk a columns paraméterben, ez a paraméter csak stringet, illetve tömböt fogad el. (Ebben a paraméterben nem használhatunk meta-azonosítókat.)

Order entitás: A megrendeléseket leíró entitás, ennek megfelelően tartalmazza a megrendelés dátumát, a szállítás állapotát (status), a megrendelő azonosítóját, illetve származtatott nem tárolt attribútumként a megrendelt DVD-ket. Az annotációi ennek az

egyedtípusnak olyan sok meglepetést már nem szolgáltatnak az előzőek ismeretében, talán csak a Select annotációban megjelenő új paraméter. Ez a paraméter az onSet paraméter, aminek a lényege a következő: a Select típusú mezőket alapértelmezésben csak lekérdezni lehet, beállítani nem, mivel ezek származtatott nem tárolt értékek, amiknek a WHERE záradékában "ki tudja milyen bonyolultságú leírások lehetnek?". Nos a fejlesztő biztosan, így jogosan merül fel az igény, hogy ha lehet bízzuk rá akkor a beállítását az ilyen mezőknek. Az onSet paraméterrel tehát egy az adott egyedben található metódus nevét kell megadni, amely metódus láthatósága hasonlóan, mint az entitás attribútumai az öröklődésben láthatónak kell lennie. (Célszerűen a seterek amúgy is publikusak szoktak lenni.) Tehát jelen esetben, ha megpróbáljuk beállítani az orderedDvds attribútum értékét, akkor a setOrderedDvds metódus kerül meghívásra.

Dvd entitás: Az alkalmazásban megrendelhető DVD-ket reprezentáló entitás. Ennek megfelelően a DVD-k fontosabb adatainak leíróit tartalmazza, a címét, a kategóriáját, a kölcsönzési árát (per nap), a kiadásának dátumát, azt hogy kik játszanak benne, illetve ha már szerepel valamilyen megrendelésben, akkor annak az azonosítóját. Az entitás annotációi közt újonnan felfedezhetjük a Nullable annotációt, ez nem definiál semmi bonyolultat, csak azt, hogy az adott mező lehet NULL értékű is az egyes előfordulásaiban. A többi mező, amelyiken ezt nem definiáljuk, azok kötelezően nem NULL értékűek kell, hogy legyenek.

Szerver Programozása:

- Perzisztencia:

A perzisztencia megvalósításáért felelős egyik legfontosabb elemet már nagyjából bemutattuk, ez nem más mint az Entity ősosztály. Természetesen ennek is vannak még olyan elemei, amelyekről nem esett szó, viszont a teljes működés bemutatása fölösleges is lenne, ugyanis nagyon sok funkciót nem használunk ki. (A lényegesebbekre pedig hangsúlyt fektettünk a példa alkalmazásban is.) Az Entity osztály felelős a saját leszármazott osztályainak tárolásáért, az adatok szinkronban tartásáért, és az adatbázis séma kialakításáért is. Ezeket a funkcióit a következő metódusaival lehet működésbe hozni:

A tárolásért felelős metódus: save(), ennek meghívásakor az Entity osztály eldönti, hogy az adott entitás éppen már egy tárolt entitás, ami már tehát létezik az adatbázisban, és annak adatait kell aktualizálni a memóriában lévővel, vagy pedig egy olyan entitásról van szó, aminek nincs megfelelője az adatbázisban. Ennek alapján pedig elvégezteti a megfelelő műveletet az entitáshoz tartozó megfelelő nevesített, vagy alapértelmezett PersistenceHandler EntityHandler-ével. (Ennek működéséről később lesz szó.)

A változások tárolásáért felelős metódus az updateFrom($old) metódus, amelynek a meghívásakor az „old” paraméterként kapott adatbázis elem leíró helyére fogja tárolni az aktuális entitás értékeit, szintén a megfelelő EntityHandler segítségével. Erre akkor lehet szükség, ha egy egyed elhagyja a szerver környezetét, és így elveszíti a közvetlen szinkront az adatbázissal. Viszont ekkor is inkább azt a módszert érdemes használni, amit mi is használunk a példa alkalmazásunkban, ennek okáról később lesz szó, amikor a konkrét példákról írunk.

Az entitás törlésért felelős metódus a remove() metódus, ennek működése elég egyértelmű, az adott entitást törli az adatbázisból a megfelelő EntitHandler segítségével. Természetesen a memóriából a konkrét példány nem törlődik.

Az entitás automatikus betöltéséért az autoLoad($id) metódus felelős, amelynek meghívásakor az „id” paraméterként kapott azonosítót reprezentáló változóhoz a konkrét példányba betöltődnek az azonosítóhoz tartozó leíró adattagok. Az Entity osztály ezt a PersistenceHandler segítségével oldja meg.

Az entitás adatbázis táblájának létrehozásáért a createTheTable() metódus

felelős, amely szintén a PersistenceHandler-től kapott EntityHandler segítségével működik. Ehhez a metódushoz létezik egy hasTableCreated() metódus is, amely értelemszerűen azt ellenőrzi le, hogy az entitáshoz létezik-e már az adatbázistáblája.

Az Entity osztály hatásköre körülbelül ezekig a funkcionalitásokig terjed ki, illetve természetesen az összes az entitást érintő leíró kezeléséig. Tehát az entitások kiválasztására már természetesen nem, kivéve ugye azt az esetet, amikor ismerjük az adott entitás pontos azonosítóját. Az entitások megkereséséért, illetve betöltéséért a PersistenceHandler osztály felelős, illetve ez az osztály fellelős az adatbázis-kapcsolatot megvalósító DataBaseConnector, illetve a specifikus, az entitások kezeléséért felelős EntityHandler tárolásáért. A PersistenceHanlder osztály egy kibővített singleton osztály, ami azt jelenti, hogy globálisan megnevezhető példányokat tartalmaz, de minden nevesített példányból pontosan csak egyet. Ennek fényében értelmet kap az, hogy egy entitáshoz létezik egy megfelelő PersistenceHandler, amelyet a globális nevével nevesít az egyedtípust leíró osztályon megadott annotáció. Viszont ennek megadása nem kötelező, elhagyásakor az alapértelmezett PersistenceHandler példánnyal fog dolgozni az adott entitás. A PersistenceHandler az egyedek keresését, illetve betöltését több módon is végezheti, de elsősorban a `selectEntities($entityClass, $where, $logicConn = "AND", $tester = "=")` metódust érdemes használni. Ez a metódus megfelelő paraméterezéssel egy entitásokból álló tömböt fog visszaadni, mégpedig azokat, amelyek megfelelnek a $where-ben megadott kritériumoknak. (És nyilván az $entityClass-ban megadott típusú a definiáló osztályuk.) A metódus megfelelő paraméterezése elég szerteágazó lehet, ugyanis a where feltételt definiálhatjuk egy egyszerű string-el, asszociatív tömbbel, illetve a leginkább ajánlott módon a Where osztály egy példányával.

A Where osztály gyakorlatilag az SQL-ben kiadható SELECT utasítás WHERE záradékát helyettesíti magas szintű környezetben. Ennek az osztálynak a bemutatására álljon itt egy forráskód részlet a példa alkalmazásból:

```
            $where = new Where("CustomerId", "=", $nOrder->customerId);
            $where->_and("Status", "=", "after")->_or("Status", "=", "need");
            $where->_and("ID", "<>", $nOrder->ID);
            return PersistenceHandler::getInstance()->selectEntities("Order",
$where);
```

Mint az jól látható a Where osztály a logikai operátoroknak megfelelő metódusokat tartalmaz, amelyek maguk is a Where osztály példányait adják vissza. A fenti

kódban azonban a logikai operátorok $where-ből, illetve egymásból való hívása nem esetleges. Ugyanis a Where osztály egy zárójelezési fát épít fel, amelynek felépítése attól függ, hogy egy-egy logikai operátor meghívásakor azt az adott operátort egy másik operátor eredményén, illetve az eredeti where példányon hívjuk. Ennek fényében a fenti mintakódból a második sorban szereplő VAGY operátor mivel egy ÉS operátor eredményén lett hívva azt fogja eredményezni, hogy az ÉS operátorban megadott logikai kifejezés, illetve a VAGY operátorban megadott logikai kifejezés zárójelezve lesz, míg a WHERE záradék többi eleme nem. Beszélvén a logikai kifejezésekről, fontos megmagyarázni a Where osztály konstruktorának paraméterezését, amely megegyezik az egyes logikai utasítások paraméterezésével. Ami pedig a következő: `__construct($field, $operator, $value1, $value2 = null)` Az első paraméter, mint a neve is mutatja az adott adatbázistáblában lévő mező neve. (Azért nem a tartalmazó objektum attribútumának neve, mert a Where példányok feloldásáért közvetlenül az EntityHandler felelős, amely már csak a saját lokális adataival dolgozhat, és semmilyen módon nem férhet hozzá közvetlenül az entitásleírókhoz. Ennek több oka is van ezek egy részéről még lesz szó.) A $operator paraméterben adhatjuk meg a megfelelő hasonlító operátort. A $value1 paraméterben megadhatjuk a hasonlítás jobboldalán álló operandust. Illetve három operandusú operátor esetén a második olyan operandust, amelyre történik a hasonlítás a $value2 segítségével adhatjuk meg.

Mindezek fényében tehát a fenti forráskódrészlet a következőképpen értelmezhető: Az alapértelmezett PersistenceHandler példánnyal szeretnénk kiválasztani azokat az Order típusú entitásokat, amelyekre igaz az a feltétel, hogy a CustomerId mezőjük egyezik a paraméterként kapott azonosítóval, és a Status mezőjük vagy "after", vagy "need" értékű, és az azonosítójuk nem egyezik a paraméterként kapott azonosítóval.

Eddig nem esett szó arról miért van szükség arra, hogy a PersistenceHandler kibővített singleton legyen, illetve, hogy konkrétan az adatbázis vezérlése milyen módokon történik. Illetve még arról se sok, hogy az egyedtípus hogyan befolyásolhatja melyik PersistenceHandler példány tartozik majd hozzá. Ezért most itt mindezekről együtt ejtünk szót, mivel az okai, illetve mikéntjei összefüggenek.

Mindenek előtt tekintsük át magát a perzisztencia-kezelés mikéntjét a PEA-ben. A magas szintű vezérlőktől az adatbázist közvetlenül vezérlő osztályokig az út körülbelül valahogy a következőképpen néz ki: Elsőként itt van a PersistenceHandler, amely magában tartalmazza a megfelelő DataBaseConnector-t és a megfelelő EntityHandlert.

Emellett létezik maga az Entity osztály ami ugye leírja az egyedeket, de közvetlen módon az adatbázissal nem kommunikál. Ez a két osztály kölcsönösen támaszkodik egymásra, ahogyan azt a fentiekben tárgyaltuk. Alapvetően viszont közvetlenül az adatbázissal egyik sem kommunikál, ennek megoldására létezik az EntityHandler, ami tulajdonképpen csak egy interfész, tehát az implementáció adatbázis típusonként eltérő lehet. Ez az oka részben annak, hogy az Entity leíróihoz nem férhet hozzá közvetlen módon. Illetve részben ez az oka annak, hogy a PersistenceHandler-ből több példányra is szükség lehet egyszerre. Viszont az EntityHandler nem felel a valódi adatbázis kapcsolatért, ennek oka az, hogy bár alapvetően adatbázis specifikus vezérlést fog valószínűleg implementálni, de semmi nem zárhatja ki annak az esélyét, hogy egyszer egy általános szabványos köztes adatbázis vezérlő kerüljön implementálásra. (Amely nagyon sok előnnyel járna.) Viszont ekkor, ha ennek az interfész implementációnak kéne implementálnia az adatbázis-kapcsolatot, akkor milyen módon tenné azt ilyen általánosított vezérlőként? Éppen ezért a kapcsolat megvalósításáért egy másik interfész a felelős, ez a DataBaseConnector ennek feladata a kapcsolatot megteremteni a megadott hálózati címen, a megadott felhasználói név, jelszó, illetve adatbázis séma név hármas segítségével. Ennek az osztálynak a példányai korlátlan számban létezhetnek, viszont ugyan ahhoz az adatbázishoz (amibe itt beleértjük a sémát is) kétszer nem érdemes, illetve nem túl kiszámítható kapcsolódni. Viszont eltérő adatbázishoz, illetve eltérő sémához természetesen lehet csatlakozni. Ez pedig önmagában magyarázza is, hogy miért van arra szükség, hogy több PersistenceHandler példány létezzen. Hiszen előfordulhat, hogy ugyanaz az alkalmazás más-más adatbázis sémán dolgozzon. Ennek fényében viszont nyilvánvaló, hogy fontos megnevezni, hogy egy-egy entitás melyik adatbázis sémához tartozik. Ennél viszont előnyösebb, ha eleve azt nevezi meg, hogy melyik PersistenceHandler-höz tartozik maga az entitás. Ezt pedig a @PersistenceHandlerName(value) annotációval lehet nevesíteni, mégpedig a value paraméterében. Az annotációt az entitás leíró osztályán lehet elhelyezni, amennyiben ez nem tesszük meg, akkor az entitás az alapértelmezett PersistenceHandler-t fogja használni.

- Session-ök:

Az adatbázissal történő kommunikációhoz hasonlóan a klienssel folytatott kommunikáció is hasonlóan magas szintű eszközökkel kell, hogy történjen, ezért a PEA-ban úgynevezett SessionPea osztályok oldják meg a kommunikációt. (Ezek hasonlóak a JAVA SessionBean-eihez.) Ezek teljesen egyszerű osztályok, amelyeken pusztán csak egy annotáció jelzi, hogy ilyen típusú.

A kommunikációt vezérlő osztályok alapvetően attól függenek, hogy a kliens böngésző alapú, vagy pedig valamilyen mobil vagy asztali kliens. Amennyiben ez utóbbiról van szó, akkor a kommunikáció első szintje a PeaConnector osztály. Ez az osztály felelős az üzenetek alapszintű felbontásáért, és értelmezéséért. Nem feladata kiszolgálni azokat, erre a SessionPeaHandler-t használja, neki pusztán annyi a feladata, hogy felbontsa az üzenetek alap struktúráját, annak megfelelően session-t hozzon létre, töröljön, vagy töltsön vissza, majd elvégezze a metódushívást a rendszerben, ami lehet példányosítás, majd a metódushívásból kapott választ a megfelelő üzenetformára csomagolva elküldje. Ha a kliens böngésző alapú, akkor ugyan kicsit bonyolultabban, de az AjaxConnector ugyan ezt a szerepet látja el, de alapvetően az AjaxConnector nem csak a SessionPeaHandlert használja, hanem sokkal inkább használja az eseménykezelést. (De erről majd később lesz szó.)

Tehát a metódushívásokért mint az jól látszik leginkább a SessionPeaHandler a felelős, így ennek az osztálynak a működéséről érdemesebb kicsit bővebben beszélni.

Egyelőre eltekintenénk attól az esettől, hogy a kliens böngésző alapú, mivel az amúgy is egy egészen külön téma lesz. Tehát az üzenetváltás valamilyen mobil vagy asztali kliens és a szerver közt történik. Elsősorban azt kell tisztázni, hogy a hívás valamilyen Stateless (állapotmentes), vagy Stateful (állapottal bíró) osztályon fog operálni, ugyanis, ha Stateless esetről beszélünk, akkor nincs szükség arra, hogy a szerveren egy új session nyíljon, illetve, hogy bármilyen állapotot tároljunk, általában ez a legritkább eset. Amennyiben viszont Stateful osztályon szeretnénk dolgozni, akkor előbb magát ezt az osztályt példányosítani kell, ekkor automatikusan létrejön egy új session, feltéve, hogy még nem volt. Innentől kezdve ez a példány ennek a sessionnek a része, és csak ebben található meg. A SessionPeaHandler képes elvégezni a példányosítást, és a session kezelését is. Mindezt tőlünk teljesen független módon végzi, így erről nekünk nem kell tudnunk. Mint említettük a metódushívások feldolgozása is a SessionPeaHandler feladata,

amelynek feldolgozásához igénybe veszi az XmlSerializer segítségét, aminek a feladata általánosságban, hogy a kommunikációban résztvevő elemek (objektumok, tömbök, egyszerű értékek, kivételek, események, illetve metódusok) XML alapú szerializációját, illetve deszerializációját megoldja.

A kommunikáció teljes működése során automatikus. Ahhoz, hogy ennek minden részletét bemutassuk, nagyon sok egyéb részletet is be kellene mutatni, amelyek most egyelőre nem tartoznak annyira szorosan a tárgyhoz. Éppen ezért inkább lényegesebbnek tartjuk bemutatni, hogy hogyan lehet definiálni egy ilyen távolról is használható osztályt, illetve hogy egy ilyen osztály milyen műveleteket végezhet. Ennek példájára álljon itt egy részlet a példa alkalmazás forráskódjából:

```
/**
 * //Annotations:
 * @Stateful
 */
class AdministratorSF {

	/**
	 *
	 * @XmlMethod(schemaName="desktop", methodName="login")
	 */
	public function login($userName, $passCode){
		$where = new Where("UserName", "=", $userName);
		$where->_and("PassCode", "=", $passCode);
		$res = PersistenceHandler::getInstance()-
>selectEntities("Administrator", $where);

		if (is_array($res) && (count($res) == 1)){
			SecureSession::getInstance()->setLoggedInUser($res[0]);
		}else{
			throw new Exception("Hibás felhasználói név, vagy jelszó.");
		}

		return "OK";
	}

	/**
	 *
	 * @param Dvd $dvd
	 * @XmlMethod(schemaName="desktop", methodName="addDvd",
paramsName={"advd" = "dvd"}, paramsType={"dvd" = "Dvd"})
	 * @GrantTo("administrator")
	 */
	public function addDvd($dvd){
		if (!$dvd->hasTableCreated()){
			$dvd->createTheTable();
		}
		$dvd->save();
	}

	/**
```

```
       *
       * @XmlMethod(schemaName="desktop", methodName="getDvdList")
       * @XmlArray(schemaName="desktop", elementName="e", valueType="Dvd")
       * @GrantTo("administrator")
       */
      public function getDvdList(){
            return PersistenceHandler::getInstance()->selectEntities("Dvd",
"1");
      }
```

Mint az jól látható ez az osztály egy Stateful osztály, ez az osztálydefiníción elhelyezett @Stateful annotációból látszik. Amennyiben ezt megadjuk, akkor a rendszer automatikusan Stateful-ként fogja kezelni az adott osztályt. Az első metódus az a login metódus, ez felel majd az adminisztrátor beléptetéséért, ezen a metóduson található egy annotáció, mégpedig a @XmlMethod nevű annotáció, amelynek két fontos paramétere van, az egyik a séma név, amely a kommunikációban használandó üzenetséma nevét jelenti. (Később erről még lesz szó.) illetve magának a metódusnak az üzenetsémában értelmezett nevét. Az XmlSerializer ennek segítségével tudja majd azonosítani a metódust, és a SessionPeaHandler ennek segítségével tudja majd meghívni az adott metódust.

A második metódus ebben a kódrészletben nem más mint az addDvd($dvd) nevű metódus. Ennél a metódusnál már kibővül az annotáció két másik paraméterrel, az egyik a paramsName, amely a metódusnak átadandó paraméterek sémabeli neveit párosítja a valódi nevükkel. (Erre, amennyiben nincs különbség nincs kötelezően szükség, viszont ebben a konkrét esetben a dvd paramétert a sémában advd-nek nevezzük, így szükséges volt ez a definíció.) Illetve a másik paraméter a paramsType nevű paraméter, ez pedig a metódus paramétereinek típusát adja meg, mégpedig a paraméter valódi nevével nevesítve. Ezt akkor szükséges megadni, ha az adott paraméter valamilyen összetett típus, ami nem egyértelműen deszerializálható. A típusok definíciójánál ki lehet hagyni azokat az elemeket amelyek deszerializációja egyértelmű, tehát nem szükséges minden paraméterhez megadni, hogy az adott paraméter milyen típusú lesz. (Ezen a metóduson látunk még egy másik annotációt is, erről a Biztonság részben lesz szó.)

A harmadik metódus a DVD-k listájának lekérdezése, amelynek a visszatérési értéke egy tömb, ennek definiálása egy fontosabb kérdés a szerializációkor, mert minden más visszatérési érték esetén vagy szerializálható osztályok példányairól van szó, vagy egyszerű primitív típusokról, viszont a tömb az egyetlen aminél ez nem ennyire triviális, főleg, hogy a szerializáló nem fog belelátni a tömbbe, hogy az milyen adatokat tartalmaz, éppen ezért az ilyen adattagok és metódusok szerializálásához szükség van a @XmlArray

annotáció megadására, amelyben definiálni kell természetesen, hogy melyik üzenetsémához tartozik az adott annotáció, azt, hogy a szerializációban a tömb elemeket milyen értékek válasszák el, illetve hogy maga a tömb milyen típusú elemeket tartalmaz. Ennek leírásával bővebb az eddigiekhez képest a harmadik metódus.

Mint az látható maguk a metódusok nem különböznek semmiben a hagyományos metódusoktól, tehát ebből is látszik, hogy a metódusokon belül nem kell semmilyen különleges dolgot tennünk, hogy működjenek a funkciók.

Fentebb említettük az üzenetsémákat, amelyek mint fő vezérszál összefüggenek a kommunikációval. Nos mivel ugyan azok az entitások, sőt akár metódusok is jelenhetnek meg különböző klienseken, így szükség volt egy olyan megoldásra, amellyel elválaszthatjuk a nagyobb adatforgalomra képes, nagyobb sávszélességű klienseket az azoknál jóval gyengébb teljesítményű társaiktól, erre pedig kézenfekvő megoldás volt, hogy az üzenetek kevésbé fontos részein az XML node-okon fogjunk, illetve a megjelenített / kommunikált attribútumok mennyiségén. Erre kifejezetten jó példa a példa alkalmazásból a DVD-ket leíró entitás sémadefiníciója:

```
$this->desktop = "<dvd>".
                        "<title>title</title>".
                        "<category>category</category>".
                        "<rentalprice>rentalPrice</rentalprice>".
                        "<releasedate>releaseDate</releasedate>".
                        "<staring>staring</staring>".
                        "<orderid>orderId</orderid>".
                  "</dvd>";

$this->mobile = "<d>".
                        "<t>title</t>".
                  "</d>";
```

Itt jól látszik, hogy a desktop nevű sémában a DVD-k minden attribútuma megtalálható, illetve minden node neve teljes, és olvasható. Ugyan ez nem szempont a mobil kliesnek esetében, akik pedig a mobile sémát használják a kommunikációra, az ő esetükben csak a DVD címe a lényeges, és nem szempont, hogy az XML olvasható legyen, így a node nevek rövidítettek. Ezzel rengeteg idő és pénz spórolható a mobil kliens oldalon.

Jelen esetben ugyan csak két séma van definiálva, de egy komolyabb alkalmazásban elképzelhető a leginkább használt készülékekhez egy-egy séma, amelynek alapján már jóval könnyebben skálázható az alkalmazás. Mivel a sémák

használata nem kötött valamilyen különleges azonosításhoz, így elképzelhető, hogy egy több asztali számítógépen futtatott kliens alkalmazás más-más üzenetsémát használjon, tehát akár ilyen módon is befolyásolható a biztonság, mivel esetleg egy titkári alkalmazásban nem látszanak az alkalmazottak fizetése, míg egy menedzseri kliensben igen. Tehát a sémáknak itt is lehet szerepük, de természetesen sokkal inkább lényeges a mobil kliensek esetében a megfelelő mennyiségű és milyenségű adatkommunikáció előállítása.

- Biztonság:

Minden webre épülő alkalmazásban a biztonság az egyik legfontosabb kérdés, természetesen éppen ezért a PEA fejlesztésénél hangsúlyt fektettünk erre is. Maga a biztonság egy szerver alkalmazás esetében nem szól másról, mint a meghívható metódusok korlátozásából azokra a személyekre, akiknek van erre joguk. A PEA-ban nincs alapértelmezett bejelentkezéskezelő, mivel ennek módja nagyban függ az alkalmazás milyenségétől is. Tehát nagy valószínűséggel nem lehet teljesen általános, mindenütt tökéletesnek mondható bejelentkeztetési rendszert alkotni. Éppen ezért ennek megvalósítása minden esetben az alkalmazás fejlesztőjének feladata. Viszont ahhoz már lehet komoly segítséget adni, hogy a korlátozásokat, illetve a jogokat deklaratív módszerekkel adjuk meg az alkalmazásban.

Az entitásoknál már volt szó a SecurityGroup-ról, ezt az interfészt implementálhatják az egyes egyedtípusok osztályai, és ezzel definiálhatják, hogy melyik biztonsági csoportba tartoznak. Ezen biztonsági csoportoknak lehet jogokat adni, illetve jogokat megtagadni az egyes metódusokra. A fentebbi kódrészletben is láttunk erre példát, a login metóduson kívül mindegyiken szerepelt a @GrantTo("administrator") ennek az annotációnak az a feladata, hogy engedélyt adjon az adott metódusra az adott biztonsági csoportba tartozó bejelentkezett felhasználónak. Ebben az annotációban az érték lehet halmaz is, így több csoportnak is adhatunk jogokat az adott metódusra. Bár a példa alkalmazásban nem használjuk ennek a metódusnak van egy párja a @ProtectFrom("csoportnév") amellyel mint a neve is mutatja megvédhetjük az adott metódust attól, hogy egy adott csoportba tartozó felhasználó operáljon rajta. Nyílván ez csak könnyebbséget jelent azokban az esetekben, amikor egy-egy metódust sok csoport használhat, viszont az összes közül csak kevés nem.

Lényeges kérdés még, hogy a rendszer honnan tudja ki van aktuálisan bejelentkezve, erre láthatunk példát a fenti kódrészletben, ide ki is emelném abból a lényeges részt:

```
SecureSession::getInstance()->setLoggedInUser($res[0]);
```

Ez a rész az összes felhasználó login metódusában meg van adva, ezzel a metódussal lehet megadni, hogy melyik felhasználó van aktuálisan bejelentkezve.

Fontos megjegyezni, hogy ez minden esetben egy session-höz tartozik, és a

bejelentkezett felhasználó csak és kizárólag a szerveren van tárolva egy változóban, így azt megváltoztatni vagy kitörölni a kliens oldalról nem lehet. Továbbá, mivel a jogok ellenőrzése közvetlenül a metódushívások módszerébe van beágyazva és kizárólag a szerveren történik, így ezt a biztonsági ellenőrzést megkerülni sem lehet. Ettől függetlenül a teljes biztonságosságot erről a módszerről sem lehet egyértelműen kijelenteni, viszont a megfelelő autentikációs módszer kialakításával elég megbízhatóan működik.

- Asztali kliens kialakítása:

Az asztali kliens esetében jóval egyszerűbb dolgunk van mint a mobil kliensek esetében, mivel sokkal magasabb absztrakció használható, jóval több lehetőség nyílik a reflexióra. Éppen ennek kihasználásával fejlesztettünk egy kliens kapcsolót JAVA SE nyelven a PEA-hoz, és ezt illetve néhány nem ennyire specifikus eszközt használva nagyon könnyen és gyorsan implementálható a kliens oldali alkalmazás. Itt elsősorban ezt a kapcsoló csomagot szeretnénk bemutatni, mivel a kliens alkalmazás minden részét bemutatni elég hosszadalmas lenne, ezért bizonyos részek bemutatását el fogjuk hagyni, és alapvetően a példa alkalmazás kódrészleteire támaszkodva fogjuk magyarázni az egyes részek működését.

Kezdjük talán az egyik legfontosabb résszel, a kapcsolat létrehozásával, és a Stateful osztály példányosításával. A kapcsolatot és a példányosítást is a Connector osztály konstruktora végzi a következő módon:

```
    private Connector() throws RemoteException {
        try {
            pc = new PeaConnector(new URL("http://localhost/Pea/peaserv.php"),
"desktop");
        } catch (MalformedURLException ex) {
        }
        dfs = new AdministratorSF(pc);
    }
```

A PeaConnector osztály feladata a kommunikáció kezelése a szerverrel, ezért ennek konstruktorában kell megadni a webcímet ahol a szervert találja, illetve meg kell adni a használni kívánt üzenetséma nevét. Ez után a PeaConnector példányt átadva az AdministratorSF kliensoldali megfelelőjének konstruktorába történik a kapcsolatnyitás, a session nyitás, és a példányosítás.

Nézzük meg magát az AdministratorSF kliens oldali megfelelőjét:

```
@Stateful
public class AdministratorSF extends Pea {

    public AdministratorSF(PeaConnector pc) throws RemoteException {
        super(pc);
    }

    @XmlMethodSchemas(@SEMethod(schemaName = "desktop", name = "login",
    parameters = {@SEAnnot(schemaName = "desktop", name = "userName"),
        @SEAnnot(schemaName = "desktop", name = "passCode")}))
```

```
    public String login(String userName, String passCode) throws RemoteException
{
        return this.methodCall("login", userName, passCode);
    }
```

Látható hogy az annotációk nagyon hasonlóak a szerveroldali annotációkhoz, tulajdonképpen ezekkel kapcsoljuk össze a szerveroldali Stateful példányt a kliensoldali Stateful példánnyal. Ami lényegesebb a kliens oldalon az az, hogy itt a Stateful és Stateless osztályoknak ki kell terjeszteniük a Pea nevű osztályt, ez fogja megoldani a metódushívásokat, illetve a példányosításokat a szerveroldalon a PeaConnector segítségével. Lényeges pont még a kódban a minden metódusban megjelenő this.methodCall("metódusnév", paraméterek) metódushívás. Ezt a metódust a Pea osztály implementálja, mégpedig úgy, hogy a megadott metódusnévvel (ami megegyezik a hívó metódus valódi kliens oldali nevével) illetve a megadott paraméterezéssel küld egy metódushívást a szerver felé. Majd a kapott választ a kliensoldali metódusunknak megfelelő alakúra alakítja, és azzal tér vissza.

Lényeges még az entitások megjelenése a kliens oldalon ennek működését is egy kódrészlettel szemléltetnénk:

```
@XmlSerializerSchemas(@SEAnnot(schemaName = "desktop", name = "dvd"))
public class DVD {


    @XmlSerializerSchemas(@SEAnnot(schemaName = "desktop", name = "title"))
    private String title;
    @XmlSerializerSchemas(@SEAnnot(schemaName = "desktop", name = "category"))
    private String category;
    @XmlSerializerSchemas(@SEAnnot(schemaName = "desktop", name =
"rentalprice"))
    private int rentalPrice;
    @XmlSerializerSchemas(@SEAnnot(schemaName = "desktop", name =
"releasedate"))
    private String releaseDate;
    @XmlSerializerSchemas(@SEAnnot(schemaName = "desktop", name = "staring"))
    private String staring;
    @XmlSerializerSchemas(@SEAnnot(schemaName = "desktop", name = "orderid"))
    private String orderId;
```

Mint az látható itt is leginkább az annotációkra kell csak hagyatkozni, az első és legfontosabb annotáció a @XmlSerializerSchemas annotáció, amelyben definiáljuk az egyes elemek megfelelő üzenetsémában való megjelenésüket, illetve azt, hogy az adott sémában milyen névvel találhatjuk meg az adott attribútumok, vagy egyedet.

A megfelelő annotációk elhelyezése után az összes, a szerveren definiált funkcionalitás elérhető lesz tökéletesen transzparens módon a kliensen is, így gyakorlatilag a kliens oldalon mindösszesen csak a megjelenítéshez szükséges elemeket kell megírni. Ezekre pedig egy sor hasznos eszköz található a példa alkalmazás forráskódjában, viszont ezek nem kapcsolódnak szorosan a témához, így itt nem mutatjuk be, viszont a forráskódban általában dokumentáltak, és elég egyértelműen használhatóak .

- Böngésző alapú kliens kialakítása:

A példa alkalmazásunkban a megrendelők esetében lényeges volt, hogy a rendszert a böngészőn keresztül érhessék el, így nincs szükségük különböző célszoftverekre, illetve Java futtatókörnyezetre, hogy használják a rendszert. De amúgy is lényeges szempont manapság az, hogy egy rendszert a felhasználók legalább egy része böngészőn keresztül érhesse el. Így itt most azt szeretnénk bemutatni, hogy a PEA segítségével milyen módon lehet ilyen alkalmazásokat fejleszteni. Előre kell azonban bocsátanunk, hogy a PEA vizuális komponensei még nem teljesek, és semmi nem garantálja, hogy a későbbiekben nem fognak megváltozni. Illetve mivel ezen a ponton jár még leginkább gyerekcipőben a fejlesztés, így nem garantált, hogy minden komponens tökéletesen és hibamentesen működik. Viszont mindezektől eltekintve egy nagyon kis szeletét szeretnénk bemutatni ennek a rétegnek is. Ahogy korábban így itt is a példa alkalmazás forráskódrészletein keresztül szeretnénk bemutatni a működést.

Először is azt kell átlátnunk, hogy a komponensek, amelyeket használni kívánunk, azok úgy kerülnek a weblapra, hogy egy javascript segítségével eseményeket illetve metódushívásokat válthatunk ki a szerver oldalon. Ezt a javascriptet természetesen tartalmazza a PEA, és az index.html oldalban ez használva is van. Ahhoz, hogy a rendszer el tudjon indulni szükség van egy belépési pontra, ez egy osztály, amelyben található egy olyan metódus, amelyet a javascript meg tud hívni közvetlenül, tehát amihez nem kellenek események. Ez a példa alkalmazásunkban a következőképpen néz ki:

```
/**
 *
 * //Annotations:
 * @AjaxStartPoint
 */
class CustomerMain {

      private $custSF;

      /**
       *
       * @return void
       * @AjaxMethod
       */
      public static function ajaxMain(){
            ...
      }
```

Amint az jól látható, ez az osztály egy teljesen közönséges osztály, azzal a

különbséggel, hogy az osztályon el van helyezve a @AjaxStartPoint annotáció, amikor a javascript meghívja az AjaxConnector-t akkor ezt az annotációt fogja keresni. Ezen túlmenően léteznie kell ebben az osztályban egy statikus metódusnak, amely annotálva van a @AjaxMethod annotációval. Ezt fogja tudni meghívni a javascript az oldal betöltődésekor, és ez lesz az aminek a feladata az egész alkalmazás elindítása lesz. Azaz ennek kell példányosítania a megfelelő komponenseket, Stateful osztályokat, és tulajdonképpen minden inicializálást ennek a metódusnak kell elvégeznie. Ennek a metódusnak a meghívása jól látszik az index.html oldal forrásában:

```
<body onload="peaMethodCall('CustomerMain', 'ajaxMain')">
```

Tehát a javascript innen tudja, hogy a meghívandó metódusnak mi a neve, illetve hogy melyik osztályban található. Ez az egyetlen hely, ahol a javascript működésével foglalkoznunk kell, innentől ugyanis eseményeken keresztül kommunikál a szerver a böngészővel. Hogy lássuk ennek működését egy-egy példát mutatunk be az események szerveroldali elkapására, illetve küldésére:

```
      /**
       *
       * @param unknown_type $event
       * @return unknown_type
       *
       * @Trigger(eventType="AjaxOnClick", senderType="PDiv",
key="orderbtnlock")
       */
      public function onOrderBtnClick($event){
            if (isset($this->custSF)){
                  $this->custSF->makeOrder();
                  $div = new PDiv("<h1>Sikeres megrendelés.</h1>");
                  $div->bind("main");
                  EventHandler::sendEvent(new AjaxExportableEvent($div));
            }else{
                  throw new Exception("A rendeléshez előbb be kell lépni.");
            }
      }
```

Ez a metódus akkor fog aktiválódni, amikor a megrendelés gombra kattint a felhasználó az oldalon, és ezzel kiváltja a gombhoz kötött AjaxOnClick eseményt. Itt ezen a metóduson található a @Trigger annotáció, ami annyit csinál, hogy a megadott típusú események, illetve megadott típusú eseményküldők, illetve megadott kulcsú zárral ellátott események esetén lefuttatja a metódust.

Eseményt pedig az EventHandler::sendEvent() metódusával lehet küldeni. A jelen

esetben egy AjaxExportableEvent-et küldünk, amiben megadjuk a fentebb létrehozott komponens példányt, amelyet bind-oltunk a main területhez az oldalon. Az AjaxExportableEvent ha nem kap plusz paramétert, akkor azt a területet, ahova bind-olva van az adott komponens, azt törli és elhelyezi rajta a komponenst. Ha megvan adva, hogy "append" módon küldje az eseményt, akkor az adott komponenst hozzáfűzi a területhez, ha pedig "refresh" móddal küldjük a komponenst, akkor megkeresi az adott területen az adott azonosítójú komponenst, és kicseréli a most küldöttre, azaz frissíti azt.

Természetesen még nagyon sok mindent lehetne leírni a böngésző alapú kliens működéséről, viszont a hely szűke miatt, és mivel ez a rész még nem annyira kiforrott, így ezt most elhagynánk, a példa alkalmazás alapján viszont el lehet indulni, mivel abban megpróbáltuk a már garantáltan működő komponenseket minél részletesebben bemutatni.

A szerver alkalmazás telepítése:

A példaalkalmazás szerverének telepítéséhez, és futtatásához szükség van az Apache2 webszerverre, a PHP5 script futtató környezetre, és a MySQL5 adatbázisrendszerre. Ezek megfelelő telepítése, és beállítása után nincs más dolgunk, mint a mintaalkalmazás szerver alkalmazását tartalmazó Pea könyvtárat az Apache által kiszolgált könyvtárba másolni (ez Unix környezetben alapbeállításokon a /var/www könyvtár.) Továbbá kialakítani egy "mrent" nevű adatbázissémát a MySQL adatbázisban, illetve erre a sémára minden jogot megadni egy "mrent" nevű felhasználónak, akinek az adatbázis jelszava: "tnerm".

Ezek után elvileg a weboldal elérhető kell legyen az adott gépen a http://localhost/Pea címen, viszont mivel feltételezhető, hogy nem ezen a gépen lesz futtatva minden alkalmazás, így ha az adott gépnek van valamilyen fix IP-címe vagy rögzített domain címe, akkor azt meg kell adni az alapértelmezésben Pea/AjaxConnection/JSLib mappában lévő peaajaxconnector.js JavaScript fájlban a következő függvényben:

```
function getAjaxConnector(){
   return "http://localhost/Pea/ajaxserv.php";
}
```

A localhost domain helyén. Illetőleg az adminisztrátor alkalmazásban is be kell állítani a szerver megfelelő elérését. Ezt a gui.Connector osztályban lehet megadni mégpedig ebben a sorban:

```
pc = new PeaConnector(new URL("http://localhost/Pea/peaserv.php"), "desktop");
```

Itt megadva a megfelelő elérési utat, majd újrafordítva az alkalmazást az adminisztrátori kliens is képes lesz kommunikálni a szerverrel.

Adminisztrátor kliens futása képekben:

Bejelentkezés:

Listázás:

Új elem hozzáadása:

Kliens Programozása

A kliens bemutatása:

- Funkcionalitás:
  - Adatkezelés
  - Kommunikáció
  - XML feldolgozás,kezelés
  - Object Factory
  - Absztrakció 1
- GUI – Grafikus megjelenítés, Szálak, Absztrakció 2
- Hibakezelés – Kommunikációs Hibák kezelése
- Telepítés

Funkcionalitás

Adatkezelés

Itt elsősorban azon adatok kezeléséről van szó, amelyek az alkalmazás működéséhez szükségesek, mint a kinézet, a beállítások, illetve a bejelentkezéshez szükségesek. Ezeket az adatokat az RMS (Record Management System) segítségével tároljuk, és operálunk ezeken. Az ehhez szükséges osztályok megtalálhatóak a *javax.microedition.rms* csomagban. A legfontosabb ezen osztályok közül a *RecordStore* osztály. Ez az osztály felelős közvetlenül az adatbázis megnyitásáért, írásáért, törléséért. Adatbázis megnyitása az openRecordStore(String recordStoreName, boolean createIfNecessary) statikus metódussal.

RecordStore rs = RecordStore.openRecordStore(myRecordStore,true);

Amennyiben a boolean érték nem true, és az aktuális adatbázis nem létezik, akkor a *RecordStoreNotFoundException* kivétel keletkezik.

Lezárni egy adatbázist a closeRecordStore() metódussal tehetjük meg.

rc.closeRecordStore();

Adatbázis törléséért a deleteRecordStore(String recordStoreName) statikus metódus felelős.

RecordStore.deleteRecordStore(myRecordStore);

Paramétere, természetesen az az adatbázis neve, amelyet törölni szeretnénk. Amennyiben az adatbázis nem létezett, akkor az imént említett kivételt kapjuk. Ha az

adatbázis nem törölhető, mert még használatban van, akkor pedig a *RecordStoreException* kivétel váltódik ki.

A lényegesebb dolgokon átfutottunk, most jöjjenek a kódrészletek, és hozzájuk a magyarázat. Először nézzük meg egy adatbázisban tárolt fájl olvasásának mikéntjét:

```
private byte []readInternal(String f) throws Exception
{
        RecordStore fr=null;
        try{
                fr =
RecordStore.openRecordStore(f,false,RecordStore.AUTHMODE_PRIVATE,false);
                RecordEnumeration re = fr.enumerateRecords(null,null,false);
                if(re.hasNextElement())
                {
                        return re.nextRecord();
                }
                return null;
        }finally{
                 try{if(fr!=null)fr.closeRecordStore();}catch(Exception e){}
        }
}
```

Ez a metódus a FireIO osztályhoz tartozik. Erről az osztályról tudni kell, hogy singleton, tehát a futás során ebből is csak egy példány fog létezni. Ez a metódus azért privát, mert tartozik az osztályhoz egy statikus metódus, ami visszatér a *readInternal* metódus visszatérési értékével. A *RecordEnumeration* interfész segítségével képes az RMS navigálni a tárolt adatok között, vagyis végigiterálhatunk az adatbázisban tárolt rekordjain, vagy azok egy részhalmazán. Az

```
enumerationRecords(null, null, false);
```

paramétereiről: az első paraméterként egy RecordFilter objektumot vár, amellyel a rekordokat szűrhetjük, null érték esetén, a mi esetünkben, az összes rekordot megkapjuk. A második paraméterként egy *RecordComparator* objektumot vár, amellyel a rekordok sorrendezését állíthatjuk be, null esetén rendezetlenül kapjuk vissza a rekordokat. Az utolsó paraméter már érdekesebb, arra szolgál, hogy ha a változás történik az adatbázisban, és ez az érték true, akkor az enumeráció automatikusan frissül. Igaz, hogy

mi nem használjuk folyton az adatbázist, de mégis mivel ez a folyamat eléggé erőforrás igényes, így mi *false* paramétert adtunk.

A metódus működése: megnyitjuk az adatbázist, majd a *nextRecord()* metódus visszaadja a soron következő elemet, amennyiben nincs ilyen akkor *null* - al tér vissza, majd lezárjuk az adatbázist.

```
    private void writeInternal(String file,byte[] buffer) throws Exception
{
        RecordStore fr=null;
        try{
            fr =
RecordStore.openRecordStore(file,true,RecordStore.AUTHMODE_PRIVATE,true);
            RecordEnumeration re = fr.enumerateRecords(null,null,false);
            if(re.hasNextElement())
            {
                int id = re.nextRecordId();
                fr.deleteRecord(id);
            }
            fr.addRecord(buffer,0,buffer.length);
        }finally{
             try{if(fr!=null)fr.closeRecordStore();}catch(Exception e){}
        }
}
```

Az írás is hasonló módon működik. Az *addRecord(byte[] data, int offset, int numBytes)* metódussal írunk, a byte sorozatot írjuk, offset indextől, numBytes darab byte-ot. A FireIO osztályban még találhatóak egyéb metódusok amelyek megkönnyítik az adatbázisok kezelését.

Kommunikáció

Hálózatkezelésről ugyan még nem beszéltünk, de mivel HTTP-kapcsolatról lesz szó, így ennek elméleti részét fölöslegesnek tartottuk kifejteni. Inkább nézzük a gyakorlatban hogyan is kell ezt megvalósítani. Implementálásra került egy *HTTPRequest* osztály, amelynek a legfontosabb részeit emeljük ki.

```
protected String baseURL = Epitomize.URL;
private HttpConnection c;
protected String dataToSend;
private String res = "";
```

Deklarációk, amik fontosak a további kód megértéséhez. Az *Epitomize* absztrakt osztályról még lesz szó bővebben, most elég annyit tudnunk róla, hogy a kliensnek van egy beállítások része, amit lementünk egy adatbázisba, a korábban említett módon, ahol be lehet állítani, többek között, az url-t ahová kapcsolódni kívánunk, és ebbe a statikus tagba tároljuk el amikor a program megkezdi működését, tehát amikor elérünk a kapcsolódáshoz, akkor ez a változó, ami jól láthatóan egy *String* változó, biztosan nem lesz *null* értékű. Azért biztosan, mert vagy korábban beolvastuk az értéket az adatbázisban tárolt rekordból, vagy ha ilyen alap tulajdonságok nincsenek beállítva, akkor a kliens alkalmazásnak egyéb funkciói nem használhatóak.

A *HTTPRequest* osztály *sendRequest()* metódusából láthatunk egy részletet:

```
c = (HttpConnection) Connector.open(baseURL, Connector.READ_WRITE);
c.setRequestMethod(HttpConnection.POST);
c.setRequestProperty("User-Agent",
      "Profile/MIDP-2.0 Configuration/CLDC-1.0");
c.setRequestProperty("Content-Type", "text/xml");
os = c.openOutputStream();
OutputStreamWriter osw = new OutputStreamWriter(os, "UTF-8");
osw.write(dataToSend, 0, dataToSend.length());
rc = c.getResponseCode();
if (rc != HttpConnection.HTTP_OK) {
   throw new Exception("HTTP response code: " + rc);
}
```

A *Connector* osztály segítségével nyithatunk meg új kapcsolatokat, ami egy Factory osztály, legfőbb feladata különféle kapcsolatokat reprezentáló objektumok létrehozása, amelyek szükség esetén konvertálhatóak (ezt láthatjuk az első sorban). A kapott paraméterek: az URL, ahová csatlakozni kívánunk, illetve a hozzáférés módja, ami READ_WRITE, merthogy írni és olvasni szeretnénk, de megadható még READ, és WRITE is, nyilván annak függvényében, hogy mire szeretnénk használni. (Az

alapértelmezett a READ_WRITE, viszont így sokkal szemléletesebb) . A következő sorban azt láthatjuk hogyan állítsuk be a HTTP-kérés módját, ami a mi estünkben a POST, az alapértelmezett a GET. A következő két sorban pedig a HTTP-kérés paramétereit állíthatjuk be, kulcs-érték formában. Kliensről lévén szó, mi kapcsolódunk a szerverhez. A következő sorban ezt láthatjuk, az *os* változó *OutputStream* típusú, ami ennek a metódusnak lokális változója, csak úgy mint a későbbiekben szereplő *is* változónak, aminek a típusa *InputStream* lesz. Ahhoz hogy tudjunk írni egy ilyen Stream-re, kell egy író, ez lesz maga az ows példány, amelynek a konstruktorába megadjuk a Stream-et, ahová írni szeretnénk, illetve megadhatjuk a kódolást is. Majd az *osw.write()* metódusát meghívva a megfelelő paraméterekkel (a string amit küldeni szeretnénk, honnan kezdve, és meddig), írunk a Stream-re. A következő sorokban, azt láthatjuk, hogy sikerült-e csatlakozni az URL-hez, amennyiben nem: dobhatunk kivételt. A Stream-ről való olvasás is hasonló módon történik, annyi különbséggel, hogy az olvasásmódja különbözhet:

```
is = c.openInputStream();
InputStreamReader iSR = new InputStreamReader(is, "UTF-8");
int len = (int) c.getLength();
if (len > 0) {
   int actual = 0;
   int bytesread = 0;
   char[] data = new char[len];
   while ((bytesread != len) && (actual != -1)) {
      actual = iSR.read(data, bytesread, len - bytesread);
      bytesread += actual;
   }
   process(data);
} else {
   int ch;
   while ((ch = iSR.read()) != -1) {
      process((byte) ch);
   }
}
```

Ez attól függ, hogy le tudjuk-e kérdezni az input hosszát, amennyiben igen, akkor a *len* változó értéke nagyobb lesz mint 0, és akkor az ott látható módon olvashatunk.

Amennyiben nem, akkor pedig az *else* ágban látható módon. Mindkettő használ egy *process* metódust. Természetesen nem ugyanazt, az első esetben a karakter tömböt használja, és egy String típusú változóba kapjuk meg az input-ot, a második esetben, pedig *byte*-ok *char*-rá *Castolása*-ként kapjuk meg az inputot. Mindkettő esetben a deklarációs részben látható *res* változónak az értékét állítja be, amelyhez tartozik egy getter (*public String getTextAnswer();*), így lekérdezhetjük az adott HTTPRequest példányhoz tartozó választ. Befejezésül annyit még, hogy ildomos a Stream-eket, és a HttpConnection - t lezárni (<példány>.*close()* metódussal).

XML feldolgozás, kezelés

Az elemző, vagy inkább feldolgozó, három osztályból áll: *XmlString2Data*, *XmlMsg*, és az *XmlNode*.

Az *XmlNode* osztály egy metódusát érdemes kiemelni, a többi funkcionalitása evidensnek tekintendő. A *getValue()* metódus működése, már nem triviális. Feltételezhetnénk, hogy egy node értékét adja vissza, ami részben igaz is, másrészről viszont ha egy node-nak találhatóak még alnode-jai, akkor ezeket az alnode-okat adja vissza String-é konvertálva. Ennek jelentőségére a későbbi példákban térünk ki.

Az *XmlMsg* osztály egy példánya lesz az egész üzenet, ami tartalmaz egyetlen *XmlNode* példányt, amelynek lesz egy neve, ez az aktuális node (Tag) – név, valamint vagy egy értéke, vagy alnode–jai, tehát további node – okról van szó, és így fog felépülni az xml.

Az *XmlString2Data* osztály az, ami az egész xml feldolgozásnak a motorja. Egyetlen metódusa publikus, mégpedig a *getXml(),* ez a metódus fogja visszaadni magát azt az Xml basenode - ot, amihez a teljes xml tartozik.
Ez a metódus meghív egy másik privát metódust, a *createXml()* - t, ami már valójában az xml generálásával törődik. Ha a hálózaton érkezett XML megsérült volna, akkor saját kivétel váltódik ki, amelynek kezeléséről a hibakezelés részben.

```
private XmlNode createXml() throws ExceptionHandler {
    XmlNode node = new XmlNode();
    node.setNodeName(lines.elementAt(szamlalo).toString());
    StringBuffer strBuff = new StringBuffer(lines.elementAt(szamlalo).toString());
    strBuff.insert(1, '/');
```

```
    for (; szamlalo < lines.size() - 1;) {
        String string = lines.elementAt(++szamlalo).toString();
        XmlNode nodeN = getNodeValue(string);
        if (nodeN == null) {
            if (string.equals(strBuff.toString())) {
                break;
            } else {
                nodeN = createXml();
            }
        }
        node.addSubNode(nodeN);
    }
    return node;
}
```

Ennek működéséhez az osztály konstruktorát kell átnéznünk. A konstruktornak van egy String típusú paramétere, ebben a paraméterben található a hálózaton érkezett karaktersorozat. Található az osztályban egy *private Vector split(String input)* specifikációjú metódus, ami annyit tesz, hogy az input karaktersorozatot feldarabolja sorokra, mégpedig egy sornak számít egy olyan karaktersorozat amelyet „><” karakterpáros zár le, viszont a sorhoz csak a „>” - jel tartozik, és a „<” jel már a következő sorhoz fog. Ha végiggondoljuk akkor látjuk, hogy vagy node-nevek, vagy komplett node-ok fognak belekerülni a *lines* vektor egy-egy indexű helyére. A *createXml()* metódus egy rekurzív metódus, első lépéseként beállítja az aktuális node nevét, majd azt mondja, hogy legyen egy *strBuff* nevű változó a lezáró tag, tehát hozzáfűz egy „/” jelet az eredeti node nevéhez. Ez a továbbiakban lesz érdekes. Indul egy ciklus ami végigmegy az egész üzeneten, ahol vesszük a rákövetkező elemet, és meghívjuk a *getNodeValue()* metódust. Ez a metódus, úgy működik, hogy vagy egy értéket ad vissza, egy node értéket, vagy pedig null-t, igazából egy részét a *keres()* metódus végzi el, amit ez a *getNodeValue()* meghív. Amennyiben *null*-t ad vissza, akkor megnézzük, hogy az imént eltárolt *strBuff* nevű változó értéke megegyezik-e az adott *lines* elem értékével, amennyiben igen akkor vége a folyamatnak, amennyiben nem, új rekurzió indul. Tehát így készül el egy XML üzenet. A programban a megfelelő helyen konkrétan így néz ki:

*XmlString2Data xmlStr2Data = **new** XmlString2Data(ret);*

*XmlMessage xmlMsg = **null**;*

*xmlMsg = **new** XmlMessage(xmlStr2Data.getXml());*

*MessageToObject msg2Obj = **new** MessageToObject(xmlMsg);*

ahol a *ret* egy *String* típusú változó, amelyben megtalálható maga, a hálózatról érkezett xml-üzenet. Ezt az *xmlMsg* változót kapja meg a következőkben említésre kerülő *MessageToObject* Object-Factory osztály.

Object Factory

Ez a *MessageToObject* osztály. Mint a nevében is benne van, ez egy olyan osztály, amely objektumokat gyárt üzenetekből, mégpedig XML üzenetekből. Mégpedig oly módon, hogy: azt az *xmlMsg* változót kapja meg konstruktorába, amelyről már beszéltünk.

*MessageToObject msg2Obj = **new** MessageToObject(xmlMsg);*

Két publikus metódussal rendelkezik ez az osztály, az egyik a *generateObject(String type)*, a másik pedig egy kereső metódus (*public static XmlNode keres(XmlNode node, String nodeName)*), ami mindössze annyit csinál, hogy a paraméterül adott XML részletben megkeresi az adott node nevét. Az osztály példányával minősítve, meghívjuk a *generateObject(String type)* metódust, ami Object típust szolgáltat vissza a hívó helyre, és a megfelelő típusúra *Castolva* megkapjuk a kívánt példányt. A *type* megmondja, hogy milyen példányt adjon vissza a generálás folyamán. Általában Vectorokat adunk vissza, amelyekben, valamilyen objektumok találhatóak. Amennyiben mégsem lista érkezik az inputról, akkor vagy egy példányt, de van olyan is amikor egy *null*-t adunk vissza, mert a funkcionalitás úgy kívánja. Egy megrendelési lista lekérdezése esetén, ez a kódrészlet fut le a *generateObject(String type)* metódusból.

```
if (type.equals("GOL")) {
    node = keres(xmlMsg.baseNode, "<result>");
    if (!node.hasMoreNodes()) {
        throw new ExceptionHandler("1Üres a lista.");
    }
    for (; node.hasMoreNodes();) {
        Order ord = new Order(node.getNextNode());
        vector.addElement(ord);
```

```
            }
            return vector;
        }
```

Amennyiben, van olyan node, hogy *<result>*, akkor kivételt dobunk, mert a lista üres. Egyébként példányosítunk egy *Order* osztályt, amely megkapja az adott xml részletet, és a felépített node-ból az adattagokat beállítja, és ezt az új Order példányt hozzáadjuk egy vectorhoz. Majd ha már a listán végigmentünk, akkor visszaadjuk ezt a vectort.
Az Order konstruktora:

```
    public Order(XmlNode xmlNode) {
        XmlNode test;
        xmlNode = xmlNode.getNextNode();
        for (int i = 0; i < xmlNode.getSubNodes().size(); i++) {
            test = (XmlNode) xmlNode.getSubNodes().elementAt(i);
            if (test.getNodeName().equals("<orid>")) {
                orderID = test.getValue();
            } else if (test.getNodeName().equals("<odvd>")) {
                test = MessageToObject.keres(test, "<d>");
                for (; test.hasMoreNodes();) {
                    Dvd dvd = new Dvd(test);
                    if (this.orderedDvdList == null) {
                        this.orderedDvdList = new Vector();
                    }
                    this.orderedDvdList.addElement(dvd);
                    test.getNextNode();
                }
            }
        }
    }
```

Majd a megfelelő helyen (a GUI - ban), ahol mindezt meghívjuk:

```
        if (selected.equals("GetOrderList")) {
            return list = new OrderList((Vector)
              msg2Obj.generateObject("GOL"));
```

```
}
```

Ahol a *list* egy *Listable* interfész típus, és a *DvdList* ezt az interfészt implementálja. A Listable interfésznek egyetlen sor specifikációja van :

```
public Vector getList();
```

Arra a részre, amikor null a visszatérési érték az Absztrakció 1 részben térünk vissza. A nem listák visszaadása is hasonlóan történik, a meghívás helyén van különbség, mert nem a *list* változó kap értéket, hanem az épp aktuálisan elvárt típusút.

Absztrakció 1

Azért ezt az alcímet kapta ez a rész, mert az *Epitomize* osztály csak egy részét, a funkcionalitáshoz tartozó részét tárgyaljuk, az Absztrakció 2 részben, pedig a megjelenítéshez tartozó részét. A funkcionalitás absztrakciójához még két másik interfész is tartozik: *com.mobirent.abstracttags* csomagban (*DataClassable, Listable*), valamint ez *Epitomize* absztrakt osztály is ebben található meg. A *DataClassable* osztály

```
public interface DataClassable {
    public String toString();
    public String reportIt();
}
```

szerepe, hogy összefogja azokat az osztályokat, amelyeket felhasználunk a kommunikáció során. A *Listable* interfészről már volt szó. Folytassuk az *Epitomize* osztállyal.

Ez az osztály fogja össze az egész klienst, ez gyakorlatilag egy kiszolgáló osztály, kevés osztály van amely nem terjeszti ki, ahol nem használjuk, ezek általában az előző két interfészt implementáló osztályok. Ez az osztály felel az adatbázisok kezeléséért, a Session-ök kezeléséért, és még sok egyébért.

Az alkalmazás működéséről már beszéltünk, most kicsit bővebben fogunk. Amikor elindítjuk, akkor első dolga az alkalmazásnak megnézni a beállításokat, vagyis az elmentett adatokat:

```
recordStoreProcesser(recordStoreReader("Adatok"));
getRecordStoreDatas();
```

Ez a három metódus az *Epitomize* absztrakt osztály implementált metódusai közt szerepel. Amelyik osztály kiterjeszti ezt az osztályt, az minősítés nélkül meg is hívhatja

ezeket. A *recordStoreReader("Adatok")* metódus semmi mást nem csinál, mint beolvas az Adatok rekordból a *FireIO* felhasználásával, és a beolvasott byte sorozatot adja vissza a *recordStoreProcesser()* metódusnak, ami egy byte tömböt vár, és egy globális Stringet feltölt a karaktersorozattá átalakított információval. A *getRecordStoreDatas()* metódus, pedig annyit csinál, hogy kiolvassa a statikus globális változókba ezeket az adatokat (például az URL-t, a felhasználó nevet, hozzá tartozó jelszót.), és így elérhetők közvetlenül ezek is. Amint a beolvasás megtörtént egyből ellenőrzésre kerülnek ezek a globális változók, ha valami hiányzik, akkor ezeket be kell állítanunk a *recordStoreWriter(String recName, String data)* segítségével.

Session-ök kezeléséért is felel. Ami nagyrészt annyit jelent a kliens oldalon, hogy a megfelelő adatokat egy Stringbe összefűzze, majd a megfelelő helyen meghívja, és mindez majd küldésre kerül.

Először is van egy Statefull osztály a szerveren, amelynek van egy *login* metódusa, amit meg kell hívnunk, hogy a szerver tudja, hogy van egy kliens, ami csatlakozni szeretne hozzá. Ha sikeres a bejelentkezés, akkor visszakapunk egy Session ID-t, ami egyedi lesz, és csak az aktuális futár telefonjához tartozik. Ez letárolódik szintén az adatbázisban. Ezzel a Session ID-vel már van jogosultsága a kliensnek metódusokat hívogatni a szerveren.

Metódushívás többféleképpen történhet:

```
public static String methodCall(String methodName, Vector paramsName, Vector paramsValue) {
    if (paramsName.size() != paramsValue.size()) {
        return null;
    }
    StringBuffer strBuff = new StringBuffer();
    for (int i = 0; i < paramsName.size(); i++) {
        if (paramsValue.elementAt(i) instanceof String) {
            strBuff.append("<" + (String) paramsName.elementAt(i) + ">" +
                    ((String) paramsValue.elementAt(i)) +
                    "</" + (String) paramsName.elementAt(i) + ">");
        } else {
            strBuff.append("<" + (String) paramsName.elementAt(i) + ">" +
                    ((DataClassable) paramsValue.elementAt(i)).reportIt() +
```

```
                "</" + (String) paramsName.elementAt(i) + ">");
        }
    }
    String msg = "<m>" +
        "<sid>" + getSessionId() + "</sid>" +
        "<oid>" + getOId() + "</oid>" +
        "<" + methodName + ">" + strBuff.toString() + "</" + methodName + ">" +
        "</m>";
    return msg;
}
```

Az első paramétere annak a metódusnak a neve, amit meg szeretnénk hívni a szerver oldalon, illetve az ehhez a metódushoz tartozó „átadni” kívánó paraméterek listája. Természetesen a paraméterek neveinek számának, és a hozzájuk tartozó értékek számának egyeznie kell. Aztán nem mást láthatunk mint ahogy egyesével összefűzzük ezeket az részeket. Fontos része az implementációnak az else ág. Amennyiben nem String típusú az adott paraméterhez tartozó érték, akkor meghívjuk a *reportIt()* metódust, ami biztosan van az adott példánynak, mivel az őt példányosító osztály implementálja a *DataClassable* interfészt. Valamint a végén láthatjuk a metódus implementációnak, ahogy ki fog nézni ez az XML, amit elküldünk a servernek. A további metódus hívások sokkal egyszerűbbek:

```
public static String methodCall(String methodName, String s) {
    String msg = "<m>" +
        "<sid>" + getSessionId() + "</sid>" +
        "<oid>" + getOId() + "</oid>" +
        "<" + methodName + ">" + s + "</" + methodName + ">" +
        "</m>";
    return msg;
}
```

További session-ökre vonatkozó metódusok:

public static *String sessionNewInst(String cName)*

Új session regisztrálása a szervernél (az a korábban említett login, amire visszakapjuk a SessionId-t vagy pedig kivételt).

public static *String sessionStop()*

Session vége. A sessiont le kell zárnunk, ha befejeztük a működést.

public static void *putSessionId2RecordStore()*

A session Id eltárolásra kerül, ennek meghívása esetén.

GUI – Grafikus megjelenítés, Szálak, Absztrakció 2

A grafikus megjelenítés a Midlet osztály segítségével történik, ami egy MIDlet. Ennek az osztálynak a startApp() metódusából részletek:

FireIO.setup(setTheme("theme.png"), 10, ***null****);*
*screen = FireScreen.getScreen(Display.getDisplay(****this****));*
*screen.setFullScreenMode(****true****);*
MainPanel mainPanel = ***null****;*
Epitomize.setScreen(screen);
*Epitomize.setMyMidlet(****this****);*
mainPanel = MainPanel.getInstance();
screen.setCurrent(mainPanel.getPanel());

Elsősorban beállítjuk a kinézetet, amit a theme.png reprezentál. A *screen* példánya a *FireScreen* osztálynak. A második sorban a kijelzőt állítjuk be, vagyis „átvesszük” a MIDlet -től. Beállítjuk a teljes képernyős módot. Aztán a kiszolgáló osztályban is beállítjuk a kijelzőt, illetve a MIDlet példányt, ami a kilépéshez szükséges. Majd példányosítjuk a *MainPanel* osztályt, és beállítjuk aktuális megjelenítésre. A *MainPanel* osztály nem terjeszti ki a FIRE *Panel* osztályát, hanem egy privát tagként létrehozzuk, és gyakorlatilag a továbbiakban minden erről a *Panel* példányról szól. Felkerülnek rá további FIRE komponensek, implementálásra kerül a hozzá tartozó *CommandListener* is, amit az *Epitomize* osztály implementál. A megjelenítésre került panel osztályok struktúrája hasonló a *MainPanel* - éhez, a meghívásuk módja különbözhet. Deklarációs rész:

private *Panel panel;*
private *Command exit =* ***new*** *Command("Kilépés", Command.EXIT, 0);*
private *Command setup =* ***new*** *Command("Beállítások", Command.BACK, 0);*

A MainPanel konstruktora

private *MainPanel() {*
panel = getMainPanel();

```
}
```

privát, mivel Singleton példányról van szó, így a futás során, egyetlen példány kerülhet előállításra.

Amivel biztosítjuk, hogy tényleg csak egyetlen példány legyen:

```
private static MainPanel INSTANCE = null;
public static MainPanel getInstance() {
    if (INSTANCE == null) {
        INSTANCE = new MainPanel();
        recordStoreProcesser(recordStoreReader("Adatok"));
        getRecordStoreDatas();
    }
    System.gc();
    return INSTANCE;
}
```

A deklarációs részben láthatjuk hogy kezdetben az *INSTANCE* változó *null*, tehát megtörténik a példányosítás, és visszaadjuk ezt a példányt. Természetesen ez a metódus nem csak erről gondoskodik. A korábban említett beállításokhoz szükséges információk kiolvasása is itt történik meg. Valamint nagyon fontos rész, ami a kódban sok helyen szerepel:

```
System.gc();
```

Ez a kis metódus a Garbage Collector-t „hívná meg”, gyakorlatilag segít felszabadítani lefoglalt memóriát. Ezt mondanom sem kell, hogy mennyire fontos mobiltelefonok esetében, igaz, a legtöbb mai telefonnak egy ekkora alkalmazás által felhasznált memória meg se kottyan, de azért nem árt optimalizálni. Nézzük csak akkor hogyan lesz példányunk:

```
INSTANCE = new MainPanel();
```

Ebben a konstruktorban, pedig egy utasítás található:

```
panel = getMainPanel();
```

Tehát a tényleges Panel példányosításáért és a komponensek elhelyezéséért ez a metódus felelős.

```
Panel main = new Panel();
main.setId("Főmenü");
```

```
        main.addCommand(exit);
        main.addCommand(setup);
        main.setCommandListener(this);
        main.add(createRow("Menü", Font.STYLE_BOLD));
        main.add(new Row());
```

A *Panel* osztályhoz hozzáadásra került egy *id* String típusú adattag, ennek a későbbiekben lesz szerepe. A rákövetkező három sorban a *main Panel* példányhoz tartozó *Command* típusú példányok hozzáadása, és a *CommandListener* hozzáadása látható. A *this* azt jelenti jelen esetben, hogy a *MainPanel* osztály implementálja a *CommandListener* interfészt, ennek egy részletét láthatjuk:

```
    public void commandAction(Command comm, Component c) {
        System.gc();
        if (comm == exit) {
            myMidlet.exit();
        }
        if (comm == setup) {
            screen.closePopup();
            screen.setCurrent(SetupPanel.getInstance().getPanel());
            return;
        }
}
```

Azért ezt a két esetet másoltuk be, mert jól látszik, hogy miért volt szükség a

```
Epitomize.setMyMidlet(this);
```

sorra, azért mert így közvetlenül elérhetjük a MIDlet-et, és használhatjuk a metódusait. A másik esetben, pedig az látható, hogy hogyan történnek a *Panel* osztályok példányosításai. A *SetupPanel* osztály is ugyanolyan felépítésű mint a *MainPanel*, kivéve természetesen, hogy milyen komponensek találhatóak rajta.

A rákövetkező sorban

```
main.add(createRow("Menü", Font.STYLE_BOLD));
```

a panelhez adunk hozzá egy *Row* konponenst, a paraméterül megkapott metódus az *Epitomize* osztály egy metódusa:

```
    public Row createRow(String text, int style) {
        Row row = new Row();
```

```
        row.setFilled(new Integer(FireScreen.defaultFilledRowColor));
        row.setBorder(true);
        row.setText(text);
        row.setFont(Font.getFont(Font.FACE_MONOSPACE, style, Font.SIZE_MEDIUM));
        row.setAlignment(FireScreen.CENTRE);
        return row;
    }
```

Amelyben létrehozunk egy *Row* példányt, amelynek megadunk egy text-et, amit megjelenít, illetve megadjuk a stílust, ahogyan megjeleníti.
A rákövetkező sorok:

```
        Row e = new Row();
        e.setLabel("Rendelések lekérdezése", FireScreen.defaultLabelFont, new
Integer(screen.getWidth() / 1), FireScreen.LEFT);
        e.setEditable(false);
        e.setIs(false);
        main.add(e);
        main.add(getChoose1("Választ", FireScreen.RIGHT));
```

Az előző CreateRow (*main.add(createRow("Menü", Font.STYLE_BOLD));*) és ez utóbbi pár sor így fog kinézni:

Itt ami fontosabb, az maga s*etIs()* metódus lenne, ezzel lehet beállítani, hogy többsorosan jelenjen meg az FString példány (ez is egy saját kis változtatás). Aztán ami még érdekesebb, az a

```
public Row getChoose1(String label, int where) {
 choose1 = new Command("", Command.OK, 0);
 Row button = new Row(label);
 button.setAlignment(where);
 button.addCommand(choose1);
 button.setCommandListener(this);
 return button;
}
```

metódus, ahol a *choose1* egy *Command* típusú változó. Azért van indexelve 1-gyel, mert három létezik belőle az *Epitomize* osztályban. Ez a metódus egy „grafikus gombot” eredményez (a fenti képen látható is, hogy épp azon a grafikus gombon van a **fókusz**). A *label* paraméter a grafikus gomb szövegét jelenti, a *where* paraméter pedig, hogy a *screen*-en hol helyezkedjen el. Gyakorlatilag ezekből az elemekből épül föl egy panel-osztály. A legtöbb általunk használt panel így épül fel, viszont van pár panel implementáció, ami az újrafelhasználhatóság tükrében készült, és egy - egy panel-osztály többféleképpen jelenik meg a kijelzőn, viszont ezekre az osztályokra is igaz, hogy Singleton - ok, és az alap felépítésük ugyanaz mint az imént említetteknél, a különbség a *Panel* - re rárakott komponensek módjában van. Ezen osztályok megírásának módjánál sokkal jobb módszerek is léteznek, csak sajnos nem a korlátozott JavaME esetében. Elsőként a beállítások résznél mutatjuk be hogy hogyan is működnek, majd a grafikus listákkal folytatjuk.

A beállításokért a *SetupPanel*, és *AllSetupPanel* felelős. Az elsőn grafikus gombokként jelennek meg az egyes beállítani kívánt „változók”, amelyeket adatbázisban tárolunk. Ezen grafikus gombok aktivizálásával egy új panel tűnik elő, amely az *AllSetupPanel* lesz. Ez a panel-osztály mindig annak függvényében változtatja a megjelenített komponenseket, hogy éppen milyen grafikus gombot aktivizálunk az *SetupPanel* - en. Az URL beállításával példálózunk, és a bármely más beállítás ugyanezen az elvi módon történik.

A SetupPanel privát getSetupPanel() metódusában:

```
pan.add(getChoose2("URL módosítása", FireScreen.CENTRE));
pan.add(new Row());
```

található ez a rész. Erről már volt szó, ez maga egy grafikus gomb. Aktivizálása esetén a következő commandAction - hez tartozó implementáció fut le:

```
if (cmd == choose2) {
   AllSetupPanel a = AllSetupPanel.getInstance();
   a.getPanel().setId("URL");
   a.idSetup();
   a.getPanel().validate();
   return;
}
```

Az első sor a szokásos Singleton példányosítás. A második sorban beállítunk egy id-t, majd a rákövetkező sorban beállítjuk a panel-t az id-nek megfelelően:

```
.
.
} else if (panel.getId().equals("URL")) {
      screen.closePopup();
      screen.setCurrent(AllSetupPanel.panel);
      row.setId("URL");
      row.setLabel("URL: ", FireScreen.defaultLabelFont, new Integer(screen.getWidth()
/ 2), FireScreen.RIGHT);
      row.setText(URL);
      panel.add(row);
      ("URL1");
   } else {
.
.
```

Ez a *row* változó egy ilyen megjelenítést eredményez:

Ha ezt aktivizáljuk, akkor megjelenik egy *TextBox*, vagyis valójában egy *RowTextBox* példány lesz példányosítva, ami kiterjeszti a *TextBox* - t, ahol is felhasználjuk a *Row* osztály egy példányának beállított id-jét (*row.setId("URL");*) a további működés végett. Amikor a *RowTextBox* példányosításra kerül, akkor a jelen példánk esetén ez a kódrészlet fog lefutni, ahol

```
    } else if (row.getId().equals("URL")) {
    super.setMaxSize(300);
    super.setString(Epitomize.URL);
}
```

a *super* minősítésű sorok, a *TextBox* konstruktorára vonatkoznak. A *RowTextBox* egy példányához két *Command* típusú változó tartozik, egy megerősítő, és egy elutasító gomb. Mindkét esetben szelektálunk, és ennek megfelelően: elutasítás esetén visszatérünk az ezt a panel-t megelőző panel megjelenítéséhez, megerősítő esetben, pedig az épp aktuálisan végrehajtandó utasításokat hajtjuk végre. URL beállítás esetén semmi más dolgunk nincs, mint az *AllSetupPanel* fenti képen látható baloldali fehér négyzetébe beleírni az imént bevitt karaktersorozatot:

```
if (this.row.getId().equals("URL")) {
    return;
}
```

Miért csak egy *return;* utasítást láthatunk? Azért mert a *commandAction* első soraiban található egy sor, ami mindig lefut:

```
row.textUpdate(getString());
```

A *row*, az az *AllSetupPanel*-en látható *row*, és ennek a kiírásra szánt karaktersorozatát beállítjuk a *RowTextBox* által kiterjesztett *TextBox*-beli karaktersorozataként. Amikor például egy lekérdezést hajtunk végre, akkor is ugyanez a folyamat meg végbe: látható egy a legutóbbi képhez hasonló beviteli sor, amit aktivizálva az imént leírt lépéseket végrehajtva annyiban fog különbözni, hogy a commandAction megfelelő részéhez, más - más utasítások hajtódnak végre. A többi beállítás pedig hasonlóképpen működik mint az eddig leírtak.

A másik osztály amiről szó lesz, az az *AnswerPanel*. Ez a Panel nem csak egy Panel, jóval több annál, a lista lekérdezésekhez ezt az osztályt használjuk. De ahhoz hogy megértsük a működését, menjünk egy kicsit vissza a *MainPanel*-hez, ahol is található egy „Rendelések lekérdezése” felirathoz tartozó „választ” grafikus gomb, amihez a

commandAction - ben ez az implementáció tartozik, tehát ha aktivizáljuk ezt a grafikus gombot, akkor ez az implementáció hajtódik végre:

```
Epitomize.threadWait();
new Thread(new Runnable() {
    public void run() {
        AnswerPanel gap = AnswerPanel.getInstance();
        gap.setLabel("Rendelések lekérdezése");
        gap.panelSetup();
        done = false;
    }
}).start();
return;
```

Elsőre ami szembetűnik, az nem más mint a névtelen szál. Használunk névtelen szálakat, mégpedig minden egyes lekérdezéshez, tehát minden egyes kapcsolódáshoz a szerverrel. Ezekre a névtelen szálakra azért van szükség, hogy ne „fagyasszuk” meg a programot, amikor egy-egy lekérdezés történik.

Az első sor semmi mást nem csinál, mint vár egy ideig, aztán a megfelelő kapcsolókat beállítja, de ezt is szálként futva. Tehát elindul a várakozó szál, majd elindul ez a névtelen szál is. Amint megkaptuk a választ, akkor egy kapcsolót false-ra állít. Amennyiben ez nem lenne false-ra állítva, akkor megkapnánk a „Kapcsolódási időtúllépés” kivételt. Erre a szálak ily módon történő kezelése végett van szükség, egy dolgot kell jól megválasztani, azt hogy mennyi időt várjunk. Ez ugye függ a hálózattól és a telefon sebességétől. Visszatérve az eredeti eszmefuttatásra: létrehozzuk az *AnswerPanel* egyetlen példányát, merthogy ez az osztály is Singleton. A példányosításban, és egy *label* beállításában semmi különleges nincs. A lényegi rész a *panelSetup()* - ban lesz:

```
public void panelSetup() {
    try {
        ht = new HTTPRequest();
        lBox.deleteAllElement();
        lBox.setPointerPos(-1);
        listPanel.removeAll();
        listPanel.setId(label);
```

```
listPanel.add(createRow(label, Font.STYLE_BOLD));
listPanel.add(lBox);
```

Létrehozunk egy HTTPRequest példányt, aminek segítségével kapcsolódunk a szerverhez. Az *lBox*

```
private ListBox lBox = new ListBox();
```

is egy Komponens, egy grafikus listát eredményez, melynek elemei a *ListElement* osztály példányai. Mivel Singleton osztályról van szó, így már lehet használtuk ezt a listát, ha még nem, akkor biztosan fogjuk még. A *setPointerPos(-1);* -re szintén azért van szükség, hogy ne az előző alkalommal használt lista aktuális pozicióján kezdjük el nézni az új listánkat. A removeAll() metódus a listPanel - en operál, ami egy Panel típusú változó, és mindent erre fogunk „felpakolni”. Beállítjuk az *Id*-jét ennek a panelnek és hozzáadunk egy szöveges *Row* - ot. Mind az id - nek, mind a Row - nak ugyanaz a karaktersorozata, amit a fenti névtelen szálba beállítottunk, és ami a fölső sorban megjelenik a *createRow* metódusnak köszönhetően. Majd hozzáadjuk ezt a grafikus listát is a *panel* - hez, ami majd végül megjelenítésre kerül.

```
Vector a = new Vector();
Vector b = new Vector();
a.addElement("sc");
b.addElement(startChar);
if (label.equals("Rendelések lekérdezése")) {
   ht.setDataToSend(Epitomize.methodCall("gol",a,b));
}
```

Létrehozunk két Vectort, amelyek az

```
Epitomize.methodCall("gol",a,b)
```

hoz szükségesek, mert a „gol” metódust fogjuk meghívni azzal az egy paraméterrel amit átadtunk. Mindezt odaadjuk a *ht* példány egy *setDataToSend* metódusának, ami beállítja

ezt a karaktersorozatot, mint elküldendő adatot. A továbbiakban hasonló if utasítások szerepelnek. Így döntjük el, hogy éppen mit szeretnénk elküldeni. A metódus vége:

```
        listFeltolt((Listable) feldolgoz(connecting(), label));
        listPanel.add(new Row());
        System.gc();
    } catch (ExceptionHandler ex) {
        screen.getCurrentPanel().showAlert(ex.getMessage(), null);
    }
```

Az első soron kívül minden világos, a kivételkezelésre még visszatérünk. Vegyük az első sort:

```
listFeltolt((Listable) feldolgoz(connecting(), label));
```

A *label* - t már tudjuk, hogy mi van benne, láttuk mire használjuk, itt is hasonló a célja. Az *AnswerPanel connecting()* metódusa, mindössze annyira hivatott, hogy elküldi az imént beállított adatot (*ht.setDataToSend(Epitomize.methodCall("gol",a,b))*), majd a választ visszaadja String-ként, természetesen. Így megkaptuk a feldolgoz metódus két paraméterét. A *feldolgoz* metódus lényegesebb része:

```
public Object feldolgoz(String answer, String selected) throws ExceptionHandler {
    String ret = "";
    ret = answer;
        XmlString2Data xmlStr2Data = new XmlString2Data(ret);
    XmlMessage xmlMsg = null;
    xmlMsg = new XmlMessage(xmlStr2Data.getXml());
    MessageToObject msg2Obj = new MessageToObject(xmlMsg);
    if (selected.equals("Rendelések lekérdezése")) {
      return list = new OrderList((Vector) msg2Obj.generateObject("GOL"));
    }
```

Itt látható az a rész, amit már hivatkoztunk, az Object Factory. Itt is több if utasítás

található, majd valamelyik illesztés során visszatérünk a megfelelő listával a *listFeltolt* első paramétereként, amit Castolunk *Listable* interfész típusúra, mivel mindegyik ilyen lista-osztály, mint a *OrderList* is, implementálja a *Listable*-t, így ez a Castolás nem okozhat gondot.

```
private void listFeltolt(Listable listable) {
    for (int i = 0; i < listable.size(); i++) {
        lBox.add(new ListElement(listable.elementAt(i).toString(), "L", false));
    }
    lBox.setBullet(FireIO.getLocalImage("box"));
    lBox.setSelectedBullet(FireIO.getLocalImage("checkedbox"));
    lBox.setCommandListener(this);
    panel.setVerticalOffset(-1);
    panel.validate();
    lBox.setPointerPos(-1);
    lBox.validate();
    System.gc();
}
```

Beállítjuk a grafikus listát, hozzáadjuk az elemeket.

Hibakezelés – Kommunikációs Hibák kezelése

A hibakezelés rész a kommunikációs hibákra vonatkozik főként, a megtévesztés elkerülése végett szerepel már az alcímben is így. Ezt természetesen saját osztállyal végezzük, ami nem több mint:

```
public class ExceptionHandler extends Exception {
    private String message = "";
    {
        System.gc();
    }
    public ExceptionHandler(String msg) {
        if (msg.equals("TCP open")) {
```

```
            message = "Mobil hálózat nem elérhető.";
            return;
        }
        if (msg.length() > 19 && msg.substring(0, 19).equals("HTTP response code:")) {
            System.out.println(msg);
            message = respondKiertekel(msg.substring(20, msg.length()));
            return;
        }
        if(msg.substring(0, 1).equals("1")){
            message = msg.substring(1, msg.length());
            return;
        }
        if(msg.endsWith("denied") || msg.endsWith("API")){
            message = "Engedélyezze az internethozzáférést!";
            return;
        }
        message = msg;
    }
```

Egy konstruktor, amelyben négy különböző kivétel kezelését tettük lehetővé, az első csak a szemléletesség végett került bele. Amikor nem elérhető a hálózat, akkor a telefon egy olyan kivételt dob, amelynek a szövege egyszerűen ennyi : „TCP open”. Amikor valami HTTP kódot kapunk vissza, akkor mindig ezt az üzenetet kapjuk : „HTTP response code:” és utána az épp aktuális kódot, amit az egyetlen metódusa ennek az osztálynak dolgoz fel.

Ez a metódusa:

```
private String respondKiertekel(String msg)
```

amelyben

```
case 500 : return "Internal Server Error!";
```

ilyen case ágak segítségével határozzuk meg az *ExceptionHandler message* privát String típusú változó értékét (500 - as hibakód esetén ezt az üzenet). A harmadik rész az általunk dobott kivételek végett került bele. Minden olyan üzenetet ide tartozik, amelyet szeretnénk megjeleníteni a kijelzőn, a felhasználó felé. Ha valamilyen hiba történt a kezelés közben,

általában felhasználói szintű hibák kezelésekor: például úgy akar regisztrálni valamit a szerver felé, hogy nem állította be a megfelelő paramétereket, akkor olyan kivételt dobunk, amelynek az első karaktere egy „1”-gyes, a további része pedig az üzenet, ahol az üzenet további részét jelenítjük meg. A negyedik rész pedig szintén a látványosság végett került bele. Az alábbi hibaüzenet a *MainPanel* - en jelenik meg abban az esetben, ha az alkalmazás elindítása esetén letiltottuk a hálózat hozzáférésének jogát, vagy pedig az alkalmazás elindítása esetén az ugyanerre a letiltásra vonatkozó kérdésre pozitív választ adunk, tehát nem engedélyezzük azt:

A „felugró” téglalap mögött álló kódrészlet:

MainPanel.getInstance().getPanel().showAlert(ex.getMessage(), ***null****);*

Ahol a *showAlert* két paramétert vár, az egyik egy karaktersorozat, a másik pedig egy *Image*. Ezzel gyakorlatilag be is fejeztük a kliens megírásának bemutatását. A lényegesebb részekre kitértünk, ami kimaradhatott, reméljük, hogy mindazok leírásával könnyen megérthető amelyeket itt közé tettünk.

Kliens telepítése

A fordítás során két fájl jelenik meg a megfelelő mappában, ha a netbeans-t beállítottuk erre.

Egy jar, és egy jad kiterjesztésű fájl. A jar (Java Archive Resource) - ban a szokásos

dolgok találhatóak, ez nem is érdekes annyira a most tárgyalni kívántak során. A jad fájl (Java Application Descriptor) kötött formában információkat tartalmaz a MIDlet - ről, így még a Jar fájl telefonra történő letöltése előtt megállapítható, hogy alkalmas-e az alkalmazás az eszközön való futásra. Nem minden esetben szükséges letölteni a Jad fájlt, de természetesen ajánlott ismerni a felépítését, ha olyan környezetbe kerülnénk, ahol nem kerül legenerálásra, vagy éppen nekünk kézzel kell módosítanunk a tartalmát. A mi esetünkben a Mobirent.jad fájl-ban ezek szerepelnek:

MIDlet-1: Midlet, , com.mobirent.gui.Midlet

Az első MIDlet neve, ikonja, és osztája a jar fájlon belül.

MIDlet-Jar-Size: 105024

A jar fájl mérete bájtokban.

MIDlet-Jar-URL: Mobirent.jar

Az URL ahonnan a jar fájlt letöltöttük.

MIDlet-Name: Mobirent

Név.

MIDlet-Permissions: javax.microedition.io.Connector.http

MIDlet - tet joggal ruházzuk fel. Vesszővel elválasztva mi is bátran bővíthetjük.

MIDlet-Vendor: Vendor

Szolgáltató szervezet.

MIDlet-Version: 1.0

Verzió.

MicroEdition-Configuration: CLDC-1.0

Konfiguráció megnevezés.

MicroEdition-Profile: MIDP-2.0

Profil megnevezés.

Mint már említettük nagyon korlátozottak egyes telefonok tárkapacitása, vagy egyszerűen csak az alkalmazásra vonatkozó korlátozások. A régebbi Nokia telefonokon tapasztalható a 40-60 kilobyte-os korlát. A mi esetünkben ez a korlát 100 Kilobyte lett. A tesztelés egy Motorola telefonon történik, amelynek szintén van egy ilyen korláta (kicsit több mint 100 kilobyte). Azok a telefonok, amelyek megfelelnek az alkalmazás rendszerkövetelményének, általában a 100 kilobyte-os alkalmazást képesek lementeni és futtatni, tehát fölötte van az alkalmazásra vonatkozó méret-korlát, feltölthetjük, futtathatjuk.

Az alkalmazás feltöltéséhez szükségünk van a gyártó által cd-n, mini cd-n, vagy egyéb módon a telefonhoz csatolt szoftverre. Ha ilyen nem lenne akkor a gyártó honlapjáról letöltjük a szoftvert, feltelepítjük. Linux alatt ezt wine-vel megtehetjük, ami annyi jelent, hogy a Windows-os futtatható alkalmazást elindítva wine-vel feltelepítjük a szoftvert (terminal-ban : wine <telepítőfájl neve>), majd ha ez megtörtént elindítjuk a szoftvert (ugyancsak wine segítségével). Aztán már a telefon tulajdonságaitól, illetve a szoftver működési elvétől függ, hogy milyen módon tudjuk telepíteni az alkalmazást. Van olyan szoftver-telefon páros, amely a jad-fájlt kéri kiválasztásra, van amelyik a jar fájlt. A megfelelőt válasszuk ki, töltsük fel. Aztán megkeressük az alkalmazást a telefonon, elindítjuk, és ha engedélyt kér az internethasználathoz (esetleg másra: telefon függő), akkor engedélyezzük ezen jogosultságokat az alkalmazás számára.

Mobilkliens Futása képekben:

Mentés, majd jelszó beállítása után: az első képen látható Panelen, a vissza - t aktiválva:


Név: Budai és Csubák
Cím: Debrecen Egyetem Tér 1
Telefonszám: 06702030401
Jelentés
[1]Kilépés
Kliens Újraindítása után
Menü
Rendelések lekérdezése
Választ
Kiszállítás megtörtént
Beállítások
[1]Kilépés
Visszavivendőek
Cím: Matrix I
[1]Kilépés
[3]Kiválaszt


Visszavivendőek
Sikeres művelet
[1]Kilépés
[3]Kiválaszt

Irodalomjegyzék

-Java ME

http://java.sun.com/javame/reference/apis.jsp

- The JAXP for J2ME

http://www.ibm.com/developerworks/wireless/library/wi-xmlparse/

- Parsing XML in J2ME

http://developers.sun.com/mobility/midp/articles/parsingxml/

-JAVA EE reference

http://java.sun.com/javaee/reference/

- MVC

http://java.sun.com/blueprints/patterns/MVC-detailed.html

- Oracle technology network

http://otn.oracle.com/

-Java ME Development Training Course

http://sw.nokia.com/id/0265fd77-4a28-4bb7-b466-8a0430651862/Java_ME_Development_Training_Course_v1_0_en.pdf

-Sun Mobile Device Technology - Introduction to Mobility Java Technology

http://developers.sun.com/mobility/getstart/

- Fire

http://sourceforge.net/projects/fire-j2me/

- PHP / PHP5:

http://www.php.net

- MySQL5:

http://www.mysql.com/

- Substance Look and Feel:

https://substance.dev.java.net/docs/getting-started.html

- Addendum Annotation support for PHP:

http://code.google.com/p/addendum/

- Ekler Péter, Forstner Bertalan, Kelényi Imre : Bevezetés a mobilprogramozásba

- Nyékiné Gaizler Judit: J2EE Útikalauz Java Programozóknak

- Lee Babin: Beginning Ajax with PHP from novice to professional

Függelék

Mobilkliens teljes aktivitás diagramja: